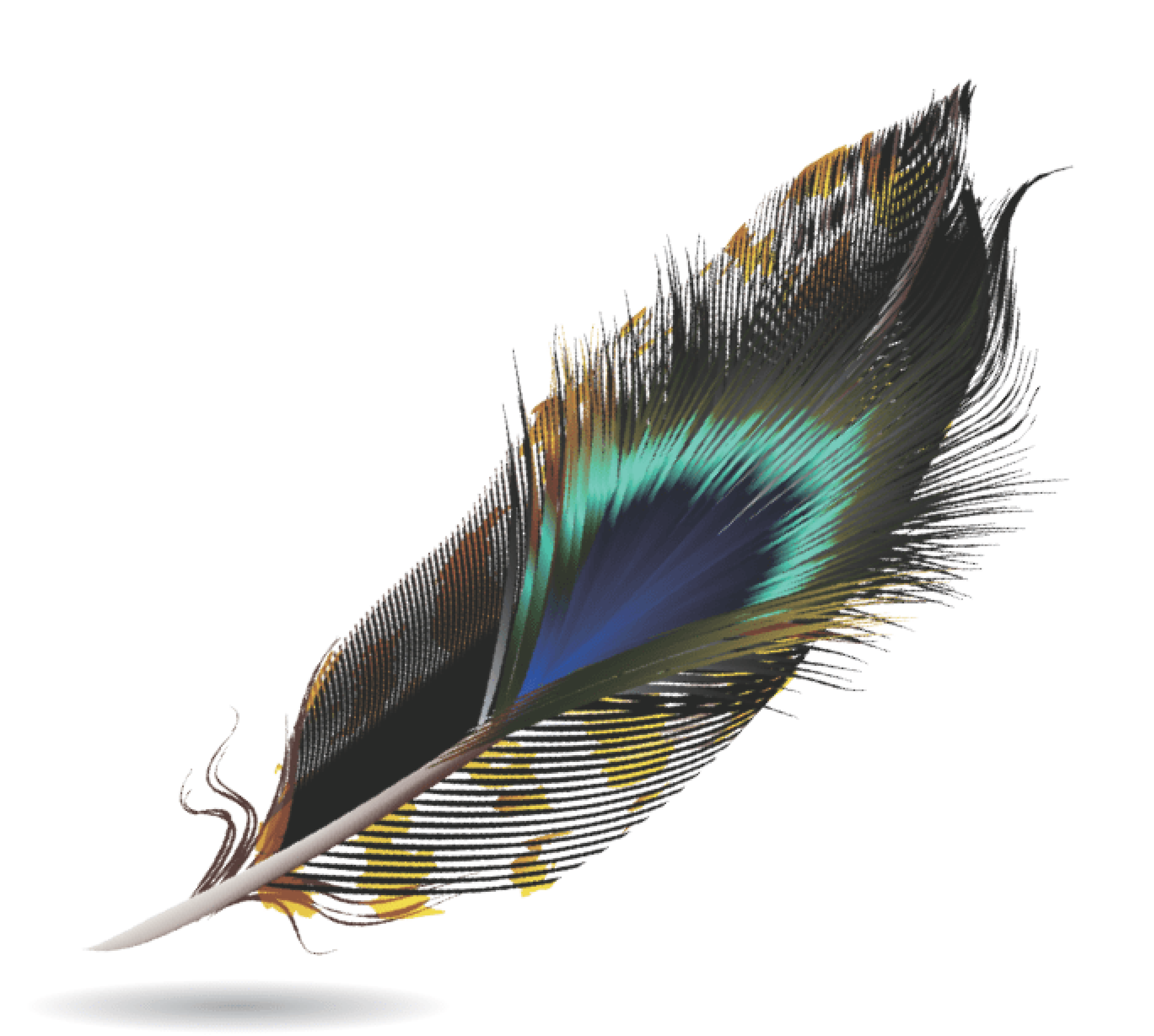


浙江省普通高校“十三五”新形态教材

大数据与人工智能技术丛书



# Python 数据分析与实践

柳毅 主编　毛峰 李艺 副主编




教学课件　教学大纲　程序源码　电子教案　习题答案

清華大学出版社

大数据与人工智能技术丛书

# Python数据分析与实践

柳　毅　主编
毛　峰　李　艺　副主编

清华大学出版社
北　京

## 内 容 简 介

Python是信息管理与信息系统、电子商务等信息管理类本科学生进行数据分析所需要掌握的基础性语言和分析工具，是未来学生掌握大数据分析技术的学习基础。本书共分12章，着重讲述Python语言和数据分析工具包的应用。第1章主要介绍Python的发展历史、特点、集成开发环境、内置模块、帮助的使用等内容；第2章主要介绍Python语言的基础知识；第3章主要介绍Python中的常用数据结构，包括序列、字典、集合等，以及函数的定义和调用等；第4章主要介绍Python中类、对象和方法的相关内容；第5章主要介绍Python进行数据分析常用的NumPy、Pandas、Matplotlib、SciPy和Scikit-learn等基础库内容；第6章主要介绍网络数据获取的HTML和XML两种网页组织形式，以及urllib和BeautifulSoup4两个模块内容；第7章主要介绍文件的操作；第8章主要介绍数据可视化，以及使用Python绘制图表的知识；第9章主要介绍利用Python进行数据库应用开发；第10、11章主要介绍Python机器学习的基本概念以及有监督、无监督学习算法的原理；第12章主要介绍Python在地理空间分析上的应用。本书中的代码均在Python 3.5中测试通过。

本书一方面侧重对Python数据分析基础知识的讲解，另一方面注重Python数据处理方法的应用。本书适合作为计算机科学与技术专业学生学习数据分析的入门教材，也适合作为Python爱好者的参考书。



**图书在版编目(CIP)数据**

Python数据分析与实践/柳毅主编. —北京：清华大学出版社，2019
(大数据与人工智能技术丛书)
ISBN 978-7-302-51579-1

Ⅰ. ①P…　Ⅱ. ①柳…　Ⅲ. ①软件工具—程序设计　Ⅳ. ①TP311.561

中国版本图书馆CIP数据核字(2018)第257419号

**策划编辑**：魏江江
**责任编辑**：王冰飞
**封面设计**：刘　键
**责任校对**：时翠兰
**责任印制**：刘海龙

**出版发行**：清华大学出版社
**网　　址**：http://www.tup.com.cn，http://www.wqbook.com
**地　　址**：北京清华大学学研大厦A座　**邮　　编**：100084
**社 总 机**：010-62770175　**邮　　购**：010-62786544
**投稿与读者服务**：010-62776969，c-service@tup.tsinghua.edu.cn
**质量反馈**：010-62772015，zhiliang@tup.tsinghua.edu.cn
**课件下载**：http://www.tup.com.cn，010-62795954
**印 装 者**：清华大学印刷厂
**经　　销**：全国新华书店
**开　　本**：185mm×260mm　**印　　张**：21.5　**字　　数**：370千字
**版　　次**：2019年8月第1版　**印　　次**：2019年8月第1次印刷
**印　　数**：1～1500
**定　　价**：59.00元

产品编号：080337-01

# 序

人类社会已经进入数字经济时代,大数据、云计算、机器学习、人工智能等技术纷至沓来,数据的管理和应用已经渗透到每一个行业的业务领域,成为当今乃至将来企业运作的基础资产。只有掌握数据并善于运用数据的人,才会在未来社会日益激烈的竞争环境中保持领先地位。

Python 语言很好地融合了大数据分析、机器学习以及人工智能技术,是目前大数据和机器学习领域热门的语言之一。本书为学习者深入浅出地介绍 Python 数据分析的原理、建模过程、统计应用方法,具有极强的实践性。

本书基于 Python 3.5 工具环境,通过实践案例讲解 Python 控制、处理、分析数据的算法和工具,让学生了解如何利用 Python 编程和数据处理库(包括 NumPy、SciPy、Matplotlib、Pandas 及 Scikit-learn 等)高效地解决各种数据分析问题,发挥 Python 在数据分析、可视化、机器学习、地理空间信息分析方面的优势,引导读者成为数据分析的高手。

本书内容严谨,逻辑清晰,可供计算机科学与技术专业以及信息管理与信息系统、电子商务等信息管理类的本科与研究生学习使用,为大数据时代的企业管理、市场营销、金融等行业从事大量数据分析的从业人员提供科学的学习资源。

浙江工业大学计算机科学与技术学院院长
国家级教学名师、教授、博士生导师
王万良

2019 年 3 月

# 前 言

Python 是大数据时代非常受欢迎的数据分析编程语言，近年来随着机器学习、云计算、人工智能等技术的发展，Python 的流行趋势扶摇直上，已经成为数据分析和数据科学事实上的标准语言和标准平台之一。

本书针对数据分析人员和 Python 编程学习者进行内容编排和章节讲述，Python 数据分析整个学习路线图计划分成 16 周，120 天左右。本书主要内容包括以下五大部分。

（1）Python 工作环境及基础语法知识：认识 Python 程序运行方式，使用 Python 3.5 开发集成环境与工具；学习 Python 程序基本结构，理解 Python 的面向对象定义和类、对象的操作方法，以及 Python 异常处理机制。本部分为基础内容，建议学习时间为 4 周。

（2）Python 数据分析相关知识：Python 生态系统为数据分析师和数据科学家提供了各种程序库，例如 NumPy、SciPy、Pandas 和 Matplotlib，使其在数据分析领域也有广泛的应用。Python 数据分析的学习主要是对相关库的使用，例如数据整理需要用到 NumPy 库，数据描述与分析则主要用到 Pandas 库。由于有前面的学习基础，本部分学习时间建议为 3 周。

（3）Python 数据可视化：Python 数据可视化的过程就是学习 Matplotlib 库的过程，Matplotlib 库包含丰富的数据可视化资源，地图、3D 等都有涉及，基于前面两部分的学习经验，这部分内容在两周内基本可以完成。

（4）Python 机器学习：Scikit-learn 是本书所使用的核心程序库，依托于上述几种工具包，封装了大量的经典以及最新的机器学习模型。通过介绍有监督和无监督机器学习原理，学习有监督学习的线性回归、Logistic 回归、朴素贝叶斯、SVM、KNN 和决策树等几个常用算法，以及无监督学习的 K-Means 聚类算法。在前面三部分学习的基础上，本部分内容建议学习时间在 4 周左右。

（5）Python 地理空间数据分析应用：向读者介绍地理空间分析的基本概念和常

用的地理空间数据，然后介绍 Python 中与地理数据处理和地理分析相关的工具，最后以 Python 处理和分析矢量数据与栅格数据的方法，对浙江省实施的“五水共治”行动中劣五类水体在地理空间模型上分布的卫星图像数据进行可视化数据分析的综合应用，本部分内容建议学习时间为 3 周。

本书编写人员具有丰富的 Python 数据分析实践经验和多年的信息管理教学能力，第 1～3 章由沈阳工业大学李艺老师编写；第 4～7、10 章由杭州电子科技大学柳毅老师编写；第 8、9、11、12 章由杭州电子科技大学毛峰老师编写；王健、陆佳涣等硕士研究生参与了本书相关章节内容和程序代码的完善工作；浙江工业大学计算机科学与技术学院院长、国家教学名师王万良教授对本书进行了认真的审阅，并提出许多宝贵的建设性意见，使本书内容日臻完善，在此对他们所付出的辛勤劳动表示诚挚的感谢。

本书结合大数据管理与应用的最新发展，针对计算机科学与技术、信息管理与信息系统、电子商务等经管类本科教学特点进行撰写。本书提供教学课件、教学大纲、电子教案、习题答案和程序源码，读者可以扫描封底的课件二维码下载。本书还提供 400 分钟的视频讲解，扫描书中的二维码，可以在线观看。

由于编者水平所限，书中难免有疏漏之处，敬请读者批评指正。

编　者

2019 年 3 月

# 目　录



# 第 1 章

# Python简介

**本章学习目标：**

- 了解 Python 语言的发展历史及特点
- 深刻了解 Python 2 和 Python 3 的区别
- 熟练掌握 Python 中 IDLE 的编程特点
- 熟练掌握 Python 的开发环境

本章首先向读者介绍 Python 发展的历史及其特点；然后分析 Python 2 与 Python 3 不同的编程特点；接着以 Eclipse＋PyDev 为例对 Python 的环境搭建进行介绍，并对 Python 自带的 IDLE 界面进行讲解；最后对 Python 的各种开发环境进行讲解。

## 1.1 Python 语言的发展史

Python 的作者是荷兰人 Guido von Rossum，尽管拥有阿姆斯特丹大学数学和计算机双硕士学位，Guido 总趋向于做计算机相关的工作，并热衷于做任何与编程相关的活儿。Guido 接触并使用过 Pascal、C、Fortran 等语言，这些语言的基本设计原则是让计算机能更快地运行。在 20 世纪 80 年代，虽然 IBM 和苹果公司已经掀起了个

人计算机浪潮，但这些个人计算机的配置很低。所有的编译器的核心是做优化，以便让程序能够运行。为了提高效率，语言也迫使程序员像计算机一样思考，以便能写出更适合计算机的程序。在那个时代，程序员恨不得拥有使用计算机每一点空间的能力，有人甚至认为C语言的指针是在浪费内存。至于动态类型、内存自动管理、面向对象等，那就不用想了，否则会让计算机陷入瘫痪。

这种编程方式让Guido感到苦恼。Guido知道如何用C语言写出一个功能，但整个编写过程需要耗费大量的时间。他的另一个选择是Shell。Bourne Shell作为UNIX系统的解释器已经长期存在，UNIX的管理员们经常用Shell去写一些简单的脚本，以进行一些系统维护的工作，比如定期备份、文件系统管理等。许多用C语言编写上百行的程序，在Shell下只用几行就可以完成。然而，Shell的本质是调用命令，它并不是一个真正的语言。比如说，Shell没有数值型的数据类型，即使加法运算也很复杂。总之，Shell不能全面地调用计算机的功能。

Guido希望有一种语言能够像C语言那样可以全面调用计算机的功能接口，又能够像Shell那样可以轻松地编程，ABC语言让Guido看到希望。ABC是由荷兰的数学和计算机研究所开发的。Guido在该研究所工作，并参与到ABC语言的开发中。与当时的大部分语言不同，ABC语言的目标是"让用户感觉更好"。ABC语言希望让程序变得容易阅读、使用、记忆和学习，并以此来激发人们学习编程的兴趣。尽管已经具备了良好的可读性和易用性，但ABC语言最终没有流行起来。在当时ABC语言的设计也存在一些致命的问题。

(1) ABC语言不是模块化语言。如果想在ABC语言中增加功能，比如对图形化的支持，就必须改动很多地方。

(2) ABC语言不能直接操作文件系统。尽管用户可以通过诸如文本流的方式导入数据，但ABC语言无法直接读写文件，输入输出的困难对于计算机语言来说是致命的。

(3) ABC语言用自然语言的方式来表达程序的意义，然而对于程序员来说，他们更习惯用function或者define来定义一个函数，用等号来分配变量。尽管ABC语言很特别，但学习难度也很大。

(4) ABC语言编译器很大，必须被保存在磁带上，这样ABC语言很难快速传播。

1989年，为了打发圣诞节假期，Guido开始写Python语言的编译器。"Python"这个名字来自Guido所挚爱的电视剧 *Monty Python's Flying Circus*。他希望

Python 语言能符合他的理想——创造一种 C 和 Shell 之间的功能全面、易学易用、可拓展的语言。1991 年，第一个 Python 编译器诞生。它是用 C 语言实现的，并能够调用 C 语言的库文件。从一诞生，Python 已经具有了类、函数、异常处理，包含表和词典在内的核心数据类型，以及以模块为基础的拓展系统。

Python 语法很多来自 C 语言，但又受到 ABC 语言的很大影响。来自 ABC 语言的一些规定直到今天还有争议，比如强制缩进，但这些语法规定让 Python 容易阅读。另外，Python 聪明地选择服从一些惯例，特别是 C 语言的惯例，比如回归等号赋值。

Python 从一开始就特别注重可拓展性，Python 可以在多个层次上拓展。用户可以直接引入. py 文件，也可以引用 C 语言的库。Python 程序员可以快速地使用 Python 写. py 文件作为拓展模块，但当性能是考虑的重要因素时，Python 程序员可以深入底层写 C 程序，编译为. so 文件引入到 Python 中使用。Python 就好像是使用钢构建房一样，先规定好大的框架，而程序员可以在此框架下自由地拓展或更改。

最初的 Python 完全由 Guido 本人开发。Python 受到 Guido 同事的欢迎，他们迅速地反馈使用意见，并参与到 Python 的改进中。Guido 和一些同事构成 Python 的核心团队，他们将自己的大部分业余时间用于 hack Python。随后，Python 被拓展到研究所之外。Python 将许多机器层面上的细节隐藏，交给编译器处理，并凸显出逻辑层面的编程思考。Python 程序员可以花更多时间用于思考程序的逻辑，而不是具体的实现细节，这一特征吸引了广大的程序员，Python 开始流行。

Guido 维护了一个邮件列表，Python 用户能通过邮件进行交流。Python 用户来自许多领域，不同的背景对 Python 也有不同的需求。Python 相当开放，又容易拓展，所以当用户不满足现有功能时很容易对 Python 进行拓展或改造。随后，这些用户将改动发给 Guido，并由 Guido 决定是否将新的特征加入到 Python 或者标准库中。如果代码能被纳入 Python 自身或者标准库，这将是极大的荣誉。由于 Guido 有着至高无上的决定权，所以他被称为“终身的仁慈独裁者”。

Python 被称为“Battery Included”，是说它的标准库功能强大。这是整个社区的贡献，Python 的开发者来自不同领域，他们将不同领域的优点带给 Python，比如 Python 标准库中的正则表达式参考 Perl，而 lambda 匿名函数以及 map()、filter()、reduce()等函数参考了 Lisp。在 Python 的开发过程中，社区起到了重要的作用。

Guido 认为自己不是全能型的程序员，所以他只负责制订框架，如果问题太复杂，他会选择绕过去，也就是 cut the corner，这些问题最终由社区中的其他人解决。社区中的人才是异常丰富的，就连创建网站、筹集基金这样与开发稍远的事情也有人乐于处理。如今的项目开发越来越复杂，越来越庞大，合作以及开放的心态成为项目最终成功的关键。从 Python 2 开始，Python 从 maillist 的开发方式转为完全开源的开发方式，社区气氛已经形成，工作被整个社区分担，Python 也获得了更加高速的发展。

到今天，Python 的框架已经确立。Python 语言以对象为核心组织代码，支持多种编程范式，采用动态类型，自动进行内存回收。Python 支持解释运行，并能调用 C 库进行拓展。Python 有强大的标准库，由于标准库的体系已经稳定，所以 Python 的生态系统开始拓展到第三方包，这些包有 Django、web. py、wxPython、NumPy、Matplotlib、PIL。

Python 崇尚优美、清晰、简单，是一种优秀并广泛使用的语言。2018 年 Python 在 TIOBE 排行榜中排行第三，它是 Google 的第三大开发语言、Dropbox 的基础语言、豆瓣的服务器语言。

Python 从其他语言中学到了很多，无论是已经进入历史的 ABC，还是依然在使用的 C 和 Perl，甚至是许多没有列出的其他语言。可以说，Python 的成功代表了它借鉴的所有语言的成功。

无论 Python 未来的命运如何，Python 的历史已经很有趣了。

### 1.1.1 Python 语言的特点

#### 1. Python 语言的优点

(1) Python 非常简单，适合阅读，阅读一个良好的 Python 程序就像是在读英语一样，尽管这个英语的要求非常严格。Python 的这种伪代码本质是它最大的优点之一，它使用户能够专注于解决问题而不是去搞明白语言本身。

(2) 易学：Python 虽然是用 C 语言写的，但是它摈弃了 C 中非常复杂的指针，简化了 Python 的语法。

(3) Python 是 FLOSS(自由/开放源代码软件)之一。简单地说，用户可以自由地发布这个软件的副本、阅读它的源代码、对它做改动，把它的一部分用于新的自由

软件中。

(4) 可移植性：由于开源本质，Python 已经被移植到许多平台上(经过改动使它能够工作在不同平台上)。如果用户小心地避免使用依赖于系统的特性，那么所有的 Python 程序无须修改就可以在 Linux、Windows、FreeBSD、Macintosh、Solaris、OS/2 以及 Google 基于 Linux 开发的 Android 平台上运行。

(5) Python 解释器把源代码转换成字节码的中间形式，然后把它翻译成计算机使用的机器语言并运行。因此，由于用户不再需要担心如何编译程序、如何确保连接转载正确的库等，所有这一切使得使用 Python 更加简单。

(6) Python 既支持面向过程的函数编程，也支持面向对象的抽象编程。在面向过程的语言中，程序是由过程或可重用代码的函数构建起来的；在面向对象的语言中，程序是由数据和功能组合而成的对象构建起来的。与其他主要的语言(例如 C++ 和 Java)相比，Python 以一种非常强大且简单的方式实现面向对象编程。

(7) 可扩展性和可嵌入性：如果用户需要让自己的一段关键代码运行得更快或者希望某些算法不公开，可以将部分程序用 C 或 C++ 编写，然后在自己的 Python 程序中使用它们。用户可以把 Python 嵌入到自己的 C/C++ 程序，从而向程序用户提供脚本功能。

(8) Python 有很庞大的标准库。它可以帮助用户处理各种工作，包括文档生成、单元测试、线程、数据库、CGI、FTP、电子邮件、XML、XML-RPC、HTML、WAV 文件、密码系统、GUI(图形用户界面)、Tk 和其他与系统有关的操作。记住，只要安装了 Python，所有这些功能都是可用的。除了标准库以外，还有许多其他高质量的库，例如 wxPython、Twisted 和 Python 图像库等。

**2. Python 语言的缺点**

(1) 在很多时候不能将 Python 程序连写成一行，例如“import sys; for i in sys.path: print i”。

(2) 运行速度：如果有速度要求，用 C++ 改写关键部分。不过对于用户而言，机器上的运行速度是可以忽略的，因为用户根本感觉不出这种速度的差异。

(3) Python 的开源性使得 Python 语言不能加密。

(4) Python 构架选择太多(没有像 C# 这样的官方.NET 构架，也没有像 Ruby

由于历史较短,构架开发的相对集中),不过这也从另一个侧面说明 Python 比较优秀,吸引的人才多,项目也多。

### 1.1.2 Python 2 与 Python 3 的区别

(1) 在 Python 3 中,print 不再是语句,而是函数,比如原来是 print 'abc' 现在是 print('abc'),但是在 Python 2.6 以上版本可以使用 from __future__ import print_function 来实现相同的功能。

(2) 在 Python 3 中没有旧式类,只有新式类,也就是说不用再像"class Foobar(object): pass"这样显式地子类化 object,但是最好加上"."，主要区别在于 old-style 是 classtype 类型,而 new-style 是 type 类型。

(3) 原来 1/2(两个整数相除)的结果是 0,现在是 0.5。Python 2.2 以上版本都可以使用 from __future__ import division 实现该特性,同时要注意"//"取代了之前的"/"。

(4) 新的字符串格式化方法 format()取代%。从 Python 2.6 以上版本开始在 str 和 unicode 中有该方法,同时 Python 3 依然支持%运算符。

(5) xrange 重命名为 range,同时更改的还有一系列内置函数及方法,它们都返回迭代器对象,而不是列表或者元组,比如 filter()、map()、dict.items 等。

(6) !=取代< >: 在 Python 2 中很少有人用< >,所以也不算什么修改。

(7) long 重命名为 int: Python 3 彻底废弃了 long+int(双整数)实现的方法,统一为 int,支持高精度整数运算。

(8) "except Exception, e"变成"except (Exception) as e": 只有 Python 2.5 及以下版本不支持该语法, Python 2.6 是支持的,不算新功能。

(9) exec 变成函数,类似 print()的变化,之前是语句。

简单补充一下:

(1) 主要是类库的变化,组织结构变了一些,但功能没变,例如 urlparse→urllib.parse 这样的变化。

(2) 对 bytes 和原生 unicode 字符串的支持,删除了 unicode 对象, str 为原生 unicode 字符串, bytes 代替了之前的 str,这是最核心的。

## 1.2　Python 的环境搭建



视频讲解

（1）进入 Python 官方网站（https://www.python.org/downloads/）下载软件包，如图 1.1 所示选择圈中区域进行下载。

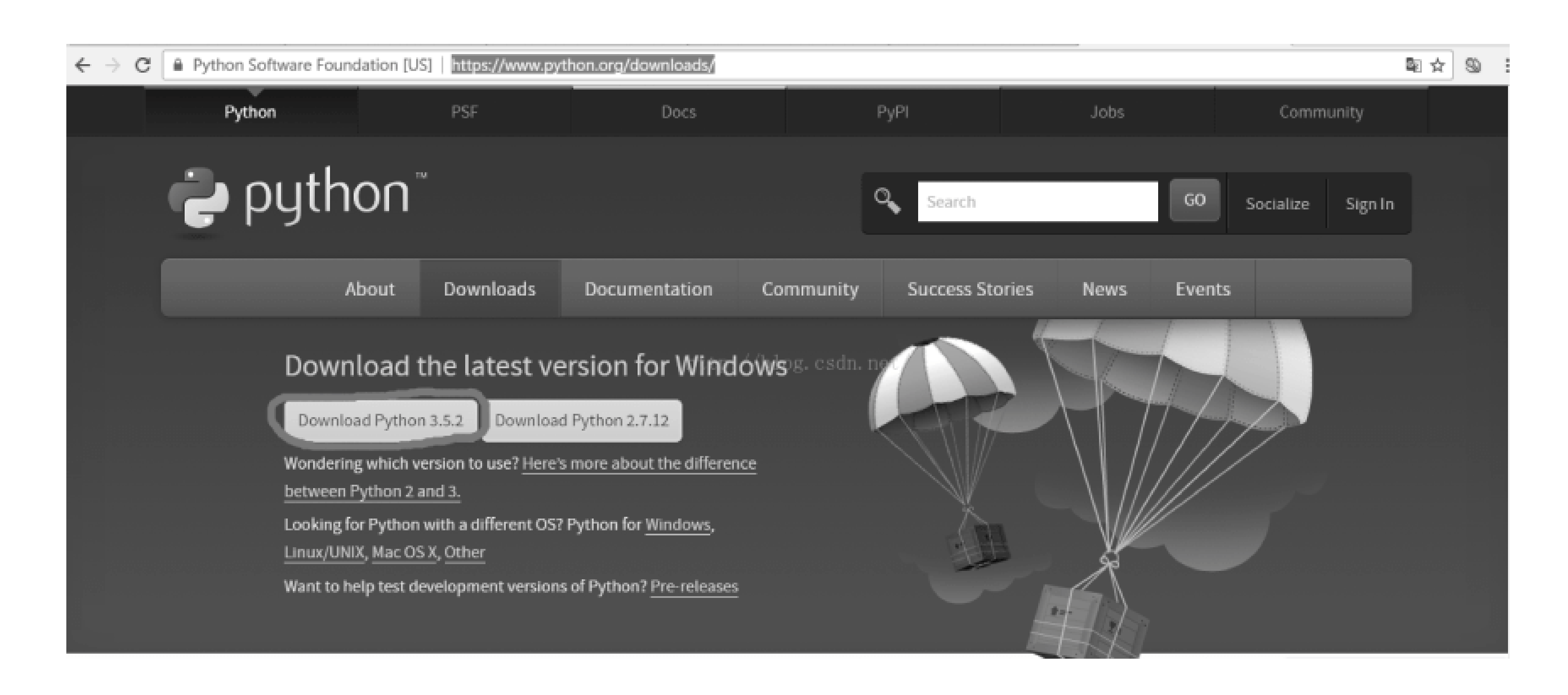



图 1.1　Python 官方网站

（2）下载完成后如图 1.2 所示。

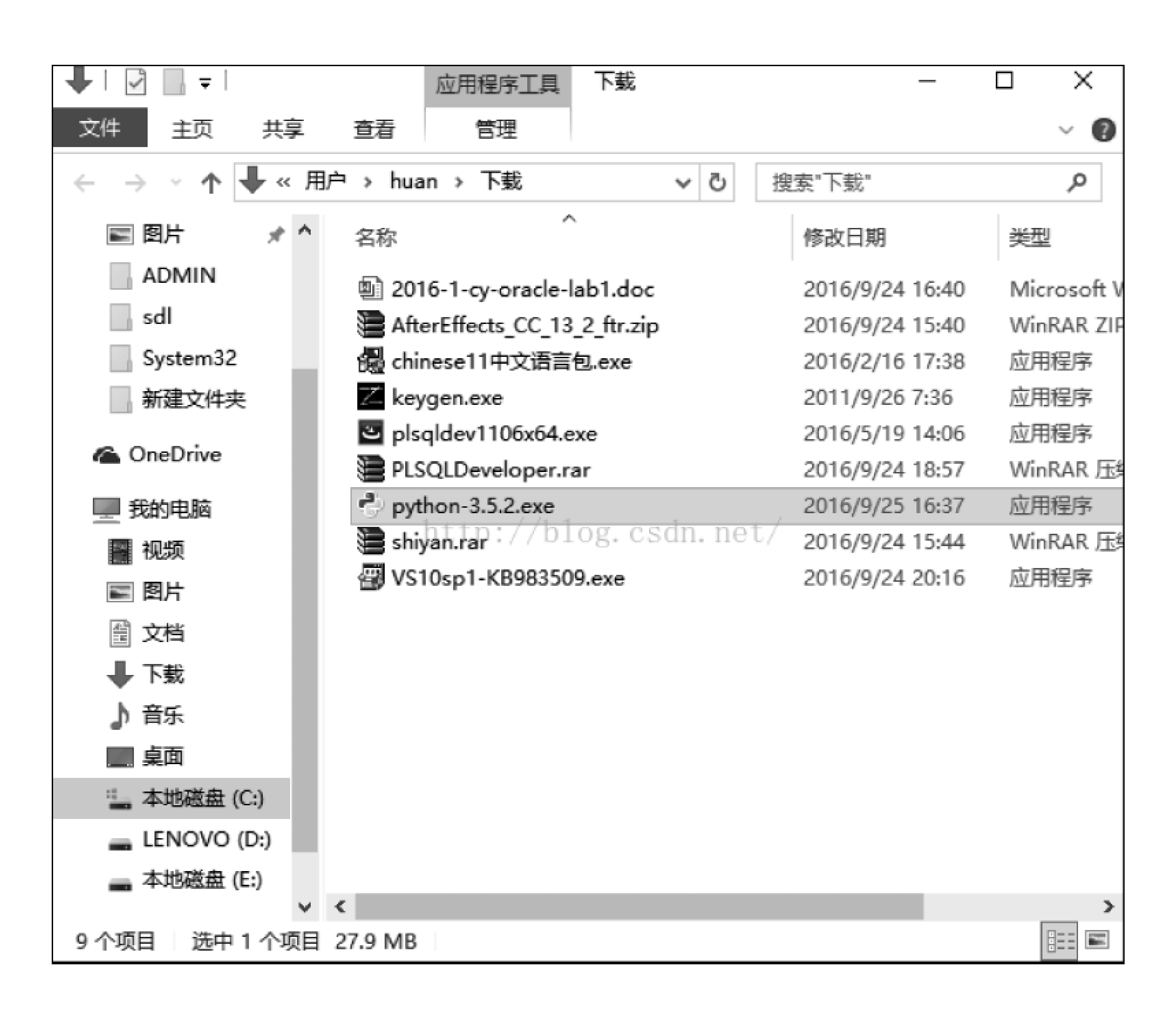



图 1.2　下载成功后的 Python 软件包

(3) 双击.exe 文件进行安装,如图 1.3 所示,并按照圈中区域进行设置,切记要勾选打钩的框,然后单击 Customize installation 进入到下一步,如图 1.4 所示。




图 1.3 Python 安装的初始化界面

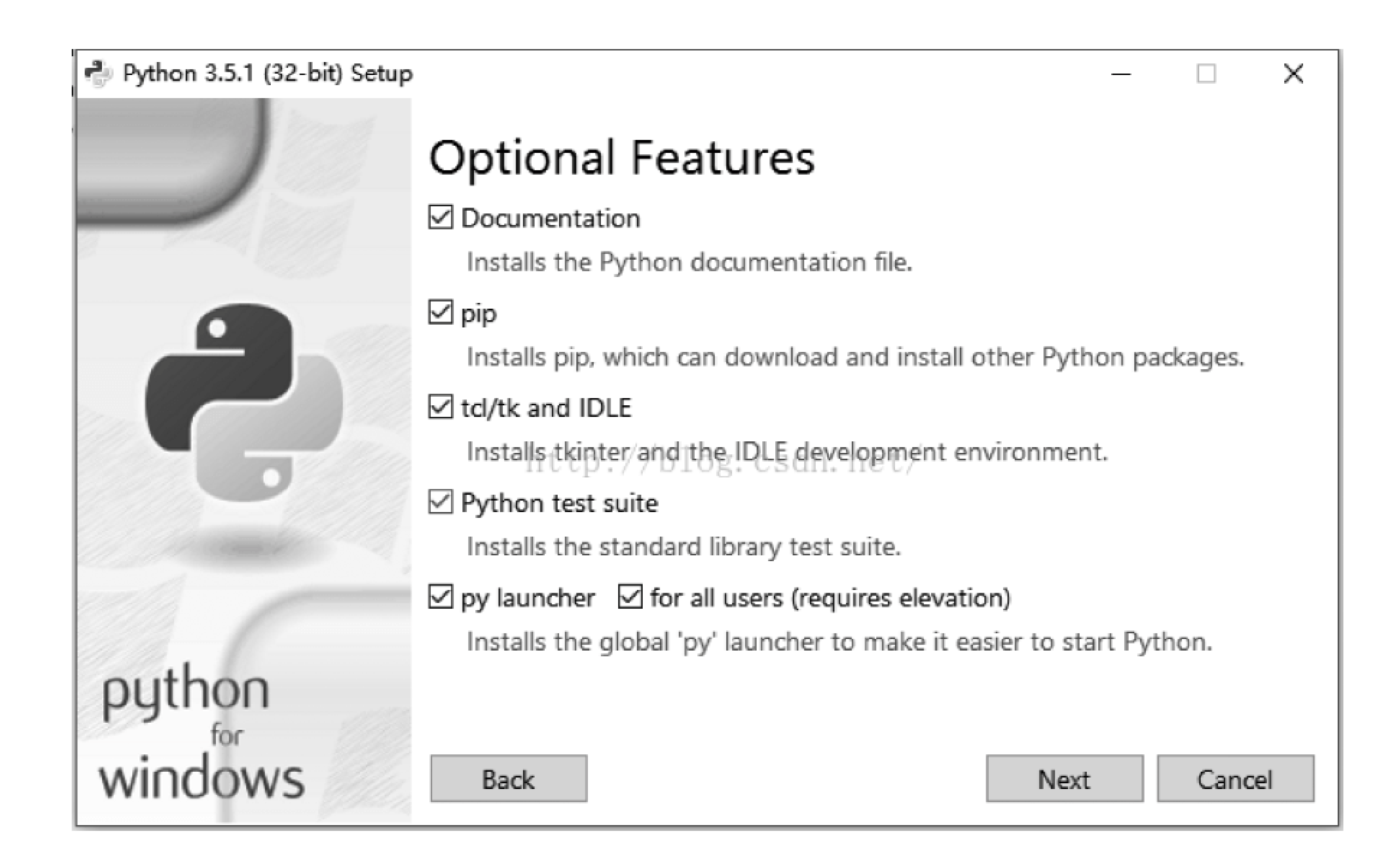



图 1.4 Python 安装过程中的配置选择

(4) 对于图 1.5,可以通过 Browse 按钮自定义安装路径,也可以直接单击 Install 按钮进行安装,单击 Install 按钮后便可以完成安装了。

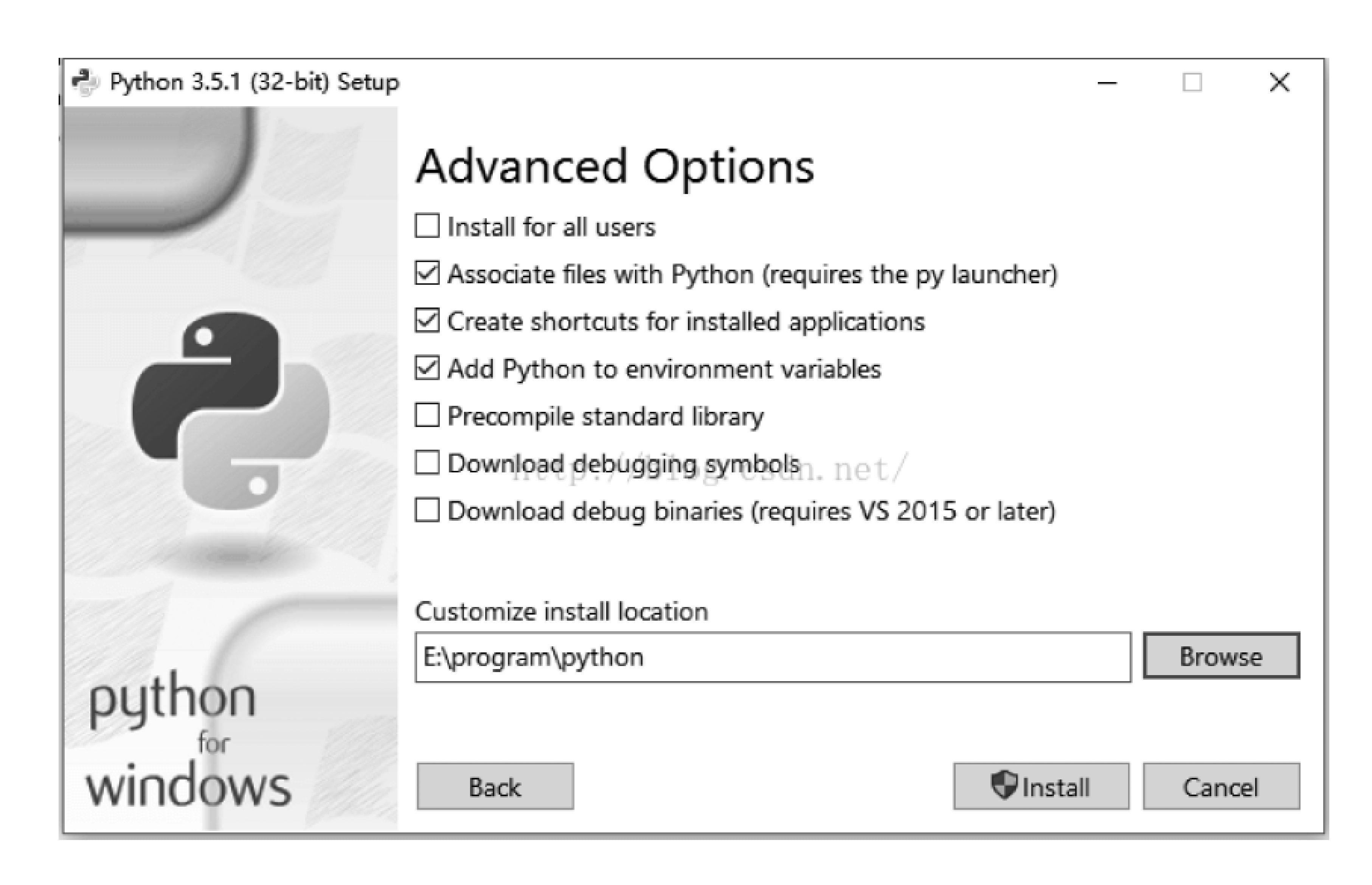



图 1.5　Python 安装过程中的路径选择

安装是很简单的，但是大家要注意 Python 要和 Java 一样配置环境变量。首先找到 Python 的安装位置，然后复制一下。

接着进入高级系统设置，单击"环境变量"按钮，在"环境变量"对话框的"系统变量"列表框中找到 Path，单击"编辑"按钮，在变量值后面添加之前复制的 Python 位置，在这前面加上英文的分号，如图 1.6 所示。

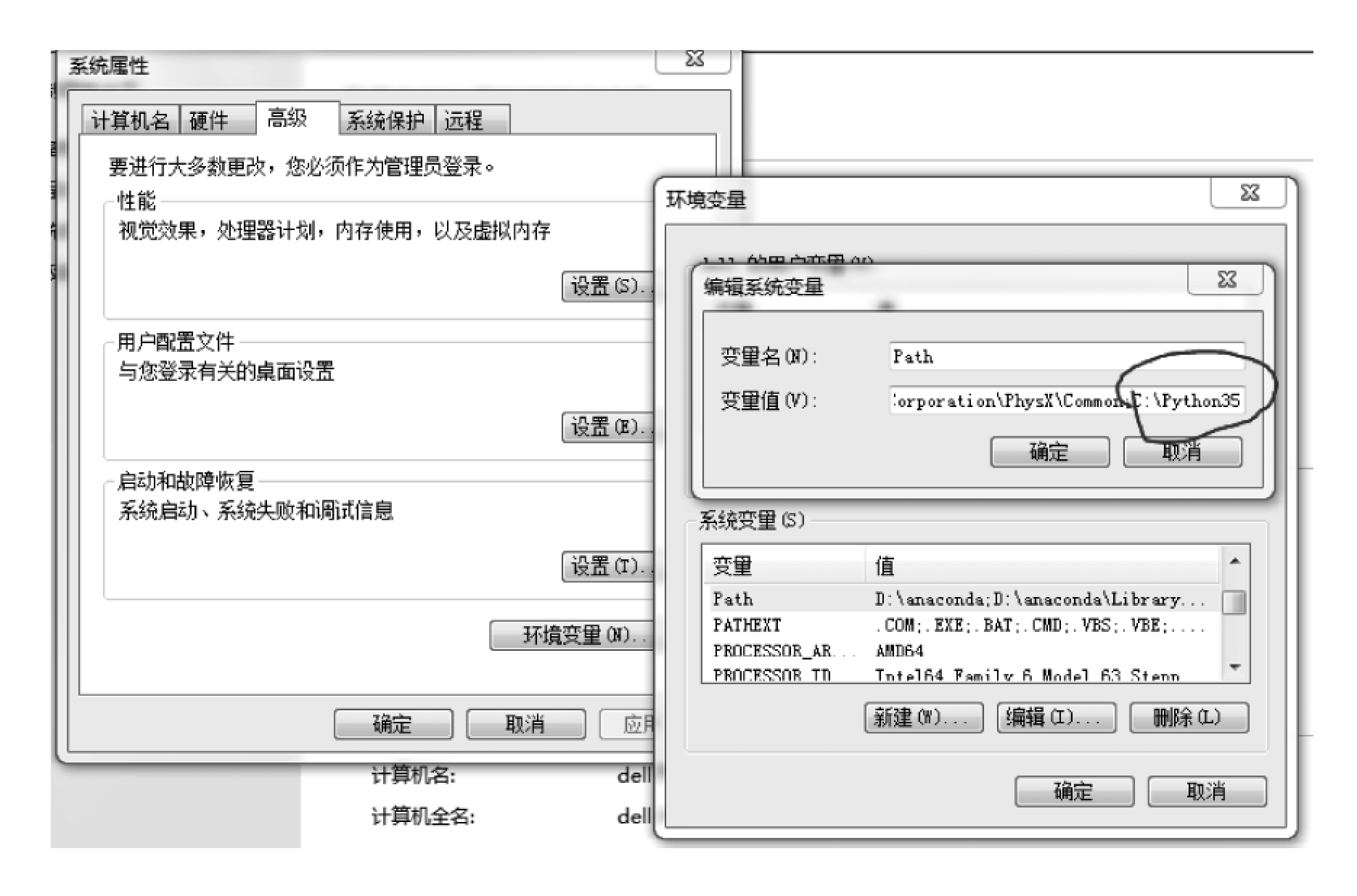



图 1.6　Python 安装中环境变量的设置

(5) 剩下的就是检验 Python 是否装好。同时按住 Win 键和 R 键,单击“确定”按钮,进入命令行,在命令行中输入 Python,出现 Python 的相关信息,则表示 Python 安装好了。

## 1.3 开始使用 Python IDLE


视频讲解

### 1.3.1 交互方式

>>>被称为 Python 命令提示符(prompt),在该提示符下,Python 正在等待用户输入代码。用户现在输入一行 Python 代码,Python 就会执行该代码。这种模式叫 Python 交互模式(interactive mode),因为 Python 在等待用户输入代码,然后执行。

例如可以输入一个表达式,让 Python 进行计算。例如要计算 1+1,可以在命令提示符后面输入 1+1,然后按 Enter 键:

```
>>> 1 + 1
```

Python 就会输出计算结果,这里是 2。如果要退出 Python 交互模式,可以在 Python 命令提示符后输入 exit()。

```
>>> exit()
```

当然也可以输入 quit()。

```
>>> quit()
```

另外,还可以输入一个 EOF(文件尾,end of file)字符,在 Windows 上是 Ctrl+Z,在 Linux 上是 Ctrl+D。对于以后的代码,如果出现以>>>开头的行,就代表这行代码是在 Python 交互模式下输入的。在 Python 交互模式下输入代码和运行.py 文件是有区别的。在 Python 命令行,Python 会等待用户一行一行地输入代码;但运行.py 文件时用户没有这个机会,而且一般运行完一个.py 文件就会立即退出(这样用户就不能看到程序输出了什么)。关于 Python 交互模式还有更多的细节,这将在以后讨论。

### 1.3.2　Python 的集成开发环境

除了从官网中下载的 Python 自带的 IDLE 之外，还有几款风格、功能各异的 IDE（集成开发环境），下面分别介绍。

#### 1. Eclipse with PyDev

其下载网址为“http://pydev.org/”，开发界面如图 1.7 所示。

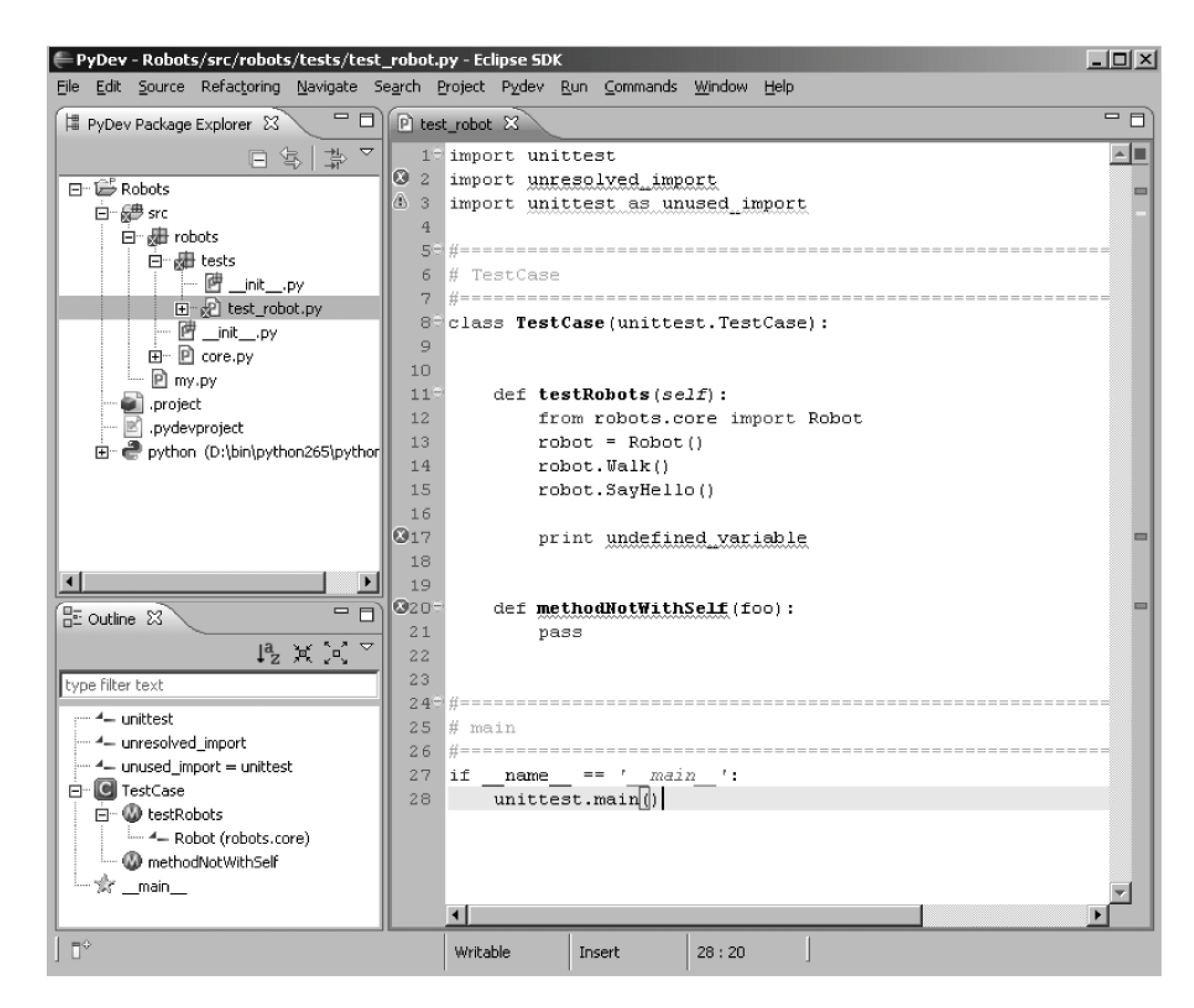



图 1.7　Eclipse with PyDev 的开发界面

Eclipse+PyDev 插件很适合开发 Python Web 应用，其特征包括代码自动完成、语法高亮显示、代码分析、调试器以及内置的交互浏览器。

#### 2. Komodo Edit

其下载网址为“http://komodoide.com/komodo-edit/”，开发界面如图 1.8 所示。

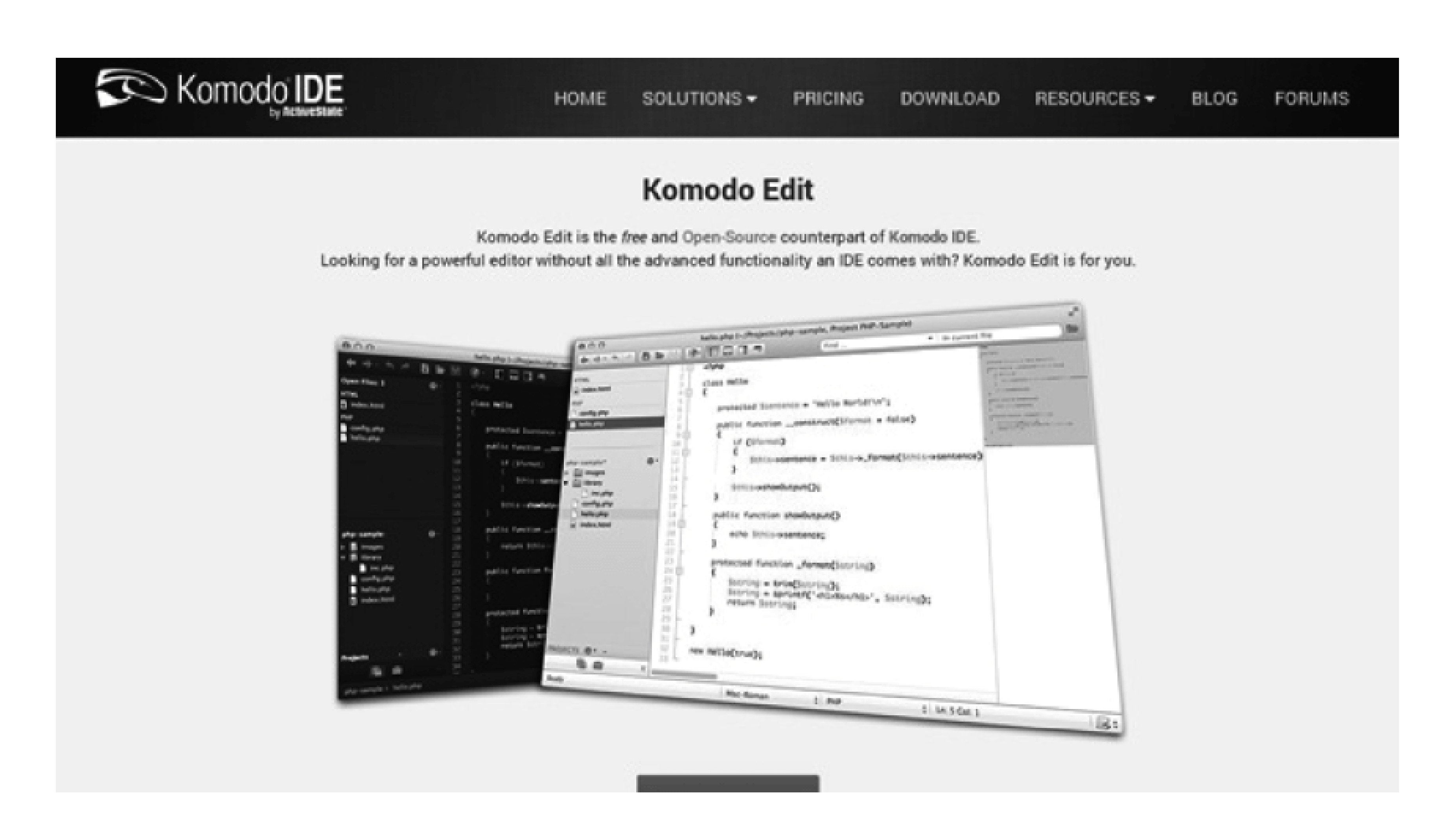



图 1.8　Komodo Edit 的开发界面

Komodo Edit 是一个免费的、开源的、专业的 Python IDE，其特征是非菜单的操作方式，开发高效。

### 3. Vim

其下载网址为“http://www.vim.org/download.php”，开发界面如图 1.9 所示。Vim 是一个简洁、高效的工具，也适合做 Python 开发。

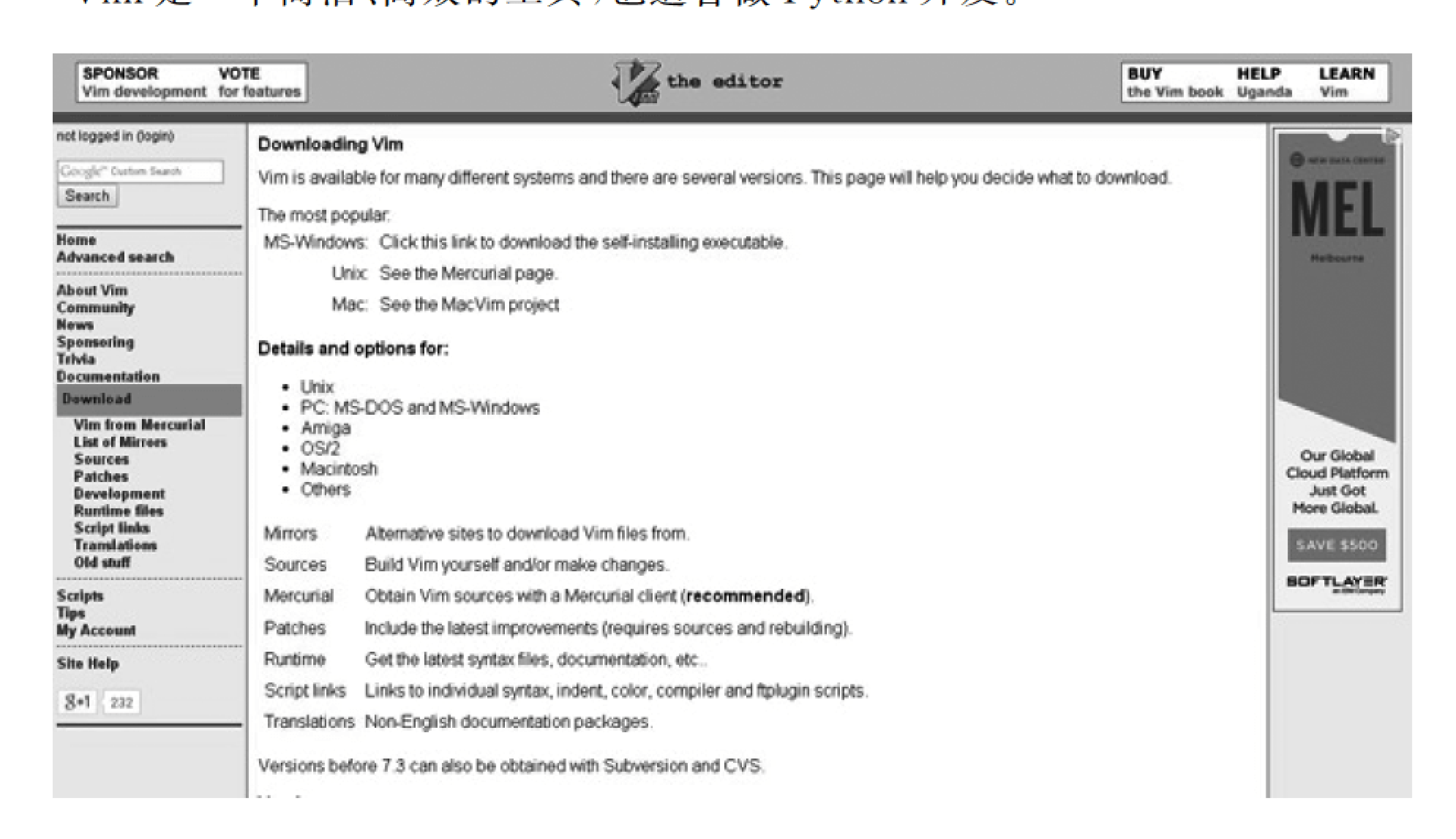



图 1.9　Vim 的开发界面

### 4. PyCharm

PyCharm 是由 JetBrains 打造的一款 Python IDE。PyCharm 具备一般 Python IDE 的功能，比如调试、语法高亮显示、项目管理、代码跳转、智能提示、代码自动完成、单元测试、版本控制等。另外，PyCharm 还提供了一些很好的功能用于 Django 开发，同时支持 Google App Engine，更酷的是 PyCharm 支持 IronPython。

PyCharm 的官方下载网址为“http://www.jetbrains.com/pycharm/download/”，开发界面如图 1.10 所示。

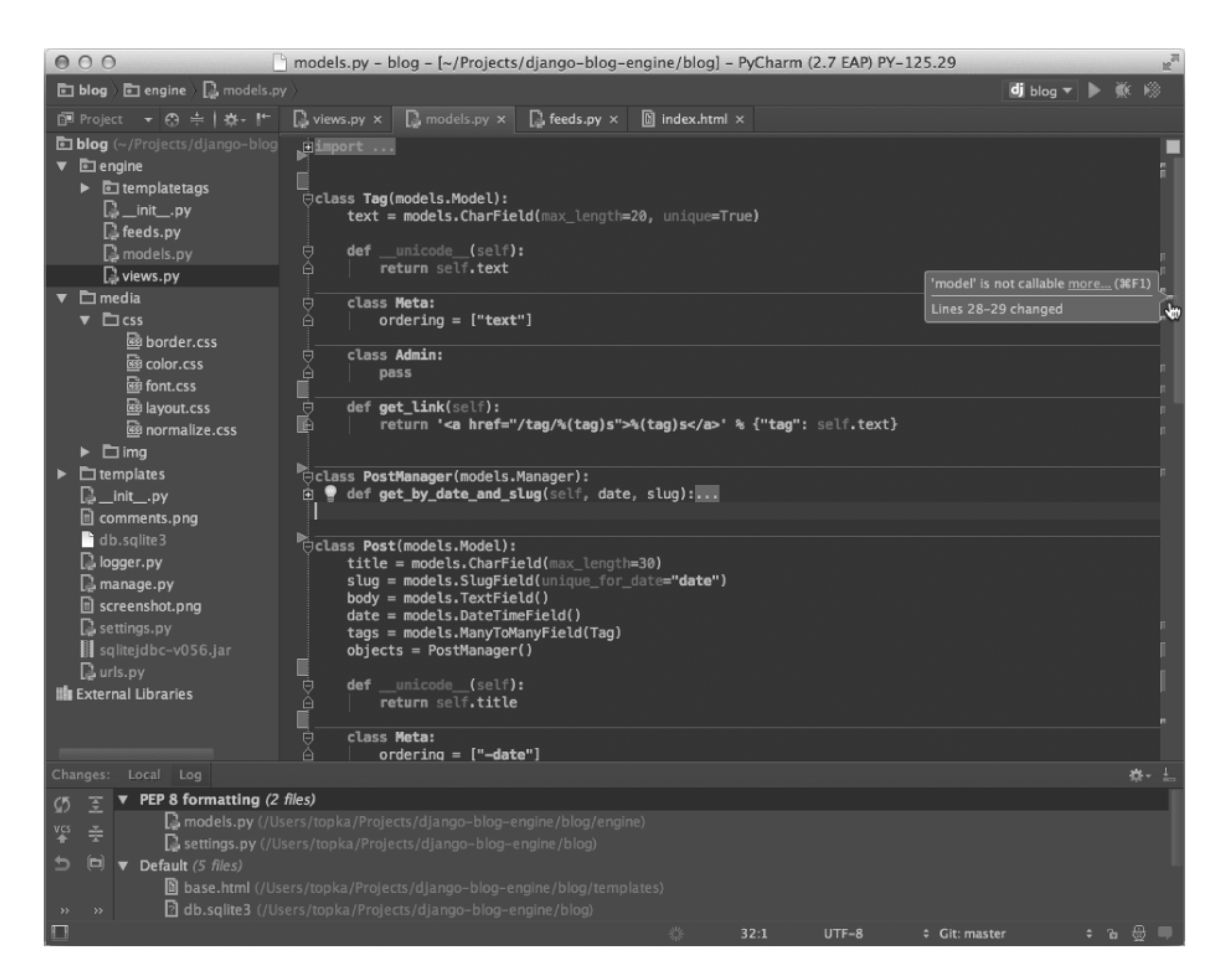



图 1.10　PyCharm 的开发界面

### 5. Sublime Text

Sublime Text 具有漂亮的开发界面（见图 1.11）和强大的功能，例如代码缩略图、Python 的插件和代码段等，还可以自定义键绑定、菜单和工具栏。Sublime Text 的主要功能包括拼写检查、书签、完整的 Python API、Goto 功能、即时项目切换、多选择、多窗口等。Sublime Text 是一个跨平台的编辑器，同时支持 Windows、Linux、

Mac OS X 等操作系统。

图 1.11 Sublime Text 的开发界面

使用 Sublime Text 2 的插件扩展功能，用户可以轻松地打造一款不错的 Python IDE，以下推荐几款插件(用户可以找到更多)。

- CodeIntel：自动补全＋成员/方法提示(强烈推荐)；
- SublimeREPL：用于运行和调试一些需要交互的程序；
- Bracket Highlighter：括号匹配及高亮显示；
- SublimeLinter：代码 pep8 格式检查。

## 1.4 Eclipse＋PyDev 的安装

### 1. 安装 Eclipse

Eclipse 可以在它的官方网站找到并下载，通常用户可以选择适合自己的 Eclipse 版本，比如 Eclipse Classic。下载完成后解压到自己想安装的目录中即可。

当然，在执行 Eclipse 之前用户必须确认安装了 Java 运行环境，即必须安装 JRE 或 JDK，用户可以到“http://www.java.com/en/download/manual.jsp”找到 JRE 下

载并安装。

### 2. 安装 PyDev

在运行 Eclipse 之后，选择 Help→Install New Software 命令，如图 1.12 所示。

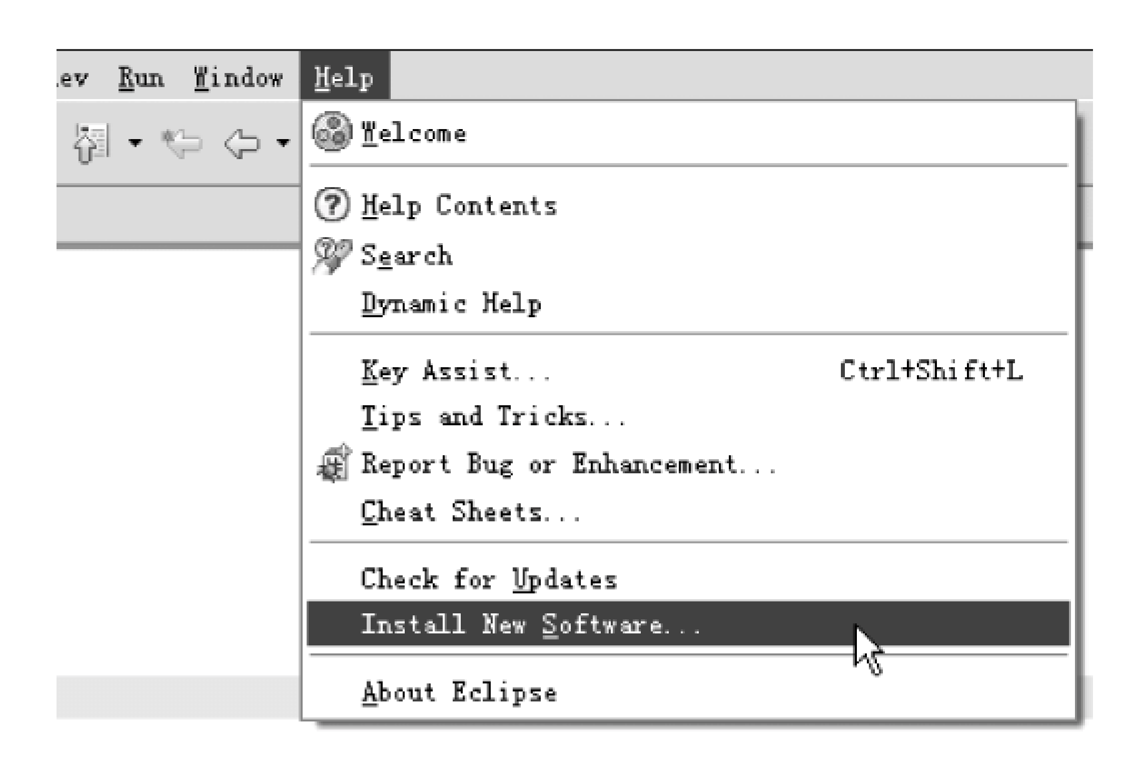



图 1.12 PyDev 的安装

单击 Add 按钮，添加 PyDev 的安装地址"http://pydev.org/updates/"，如图 1.13所示。

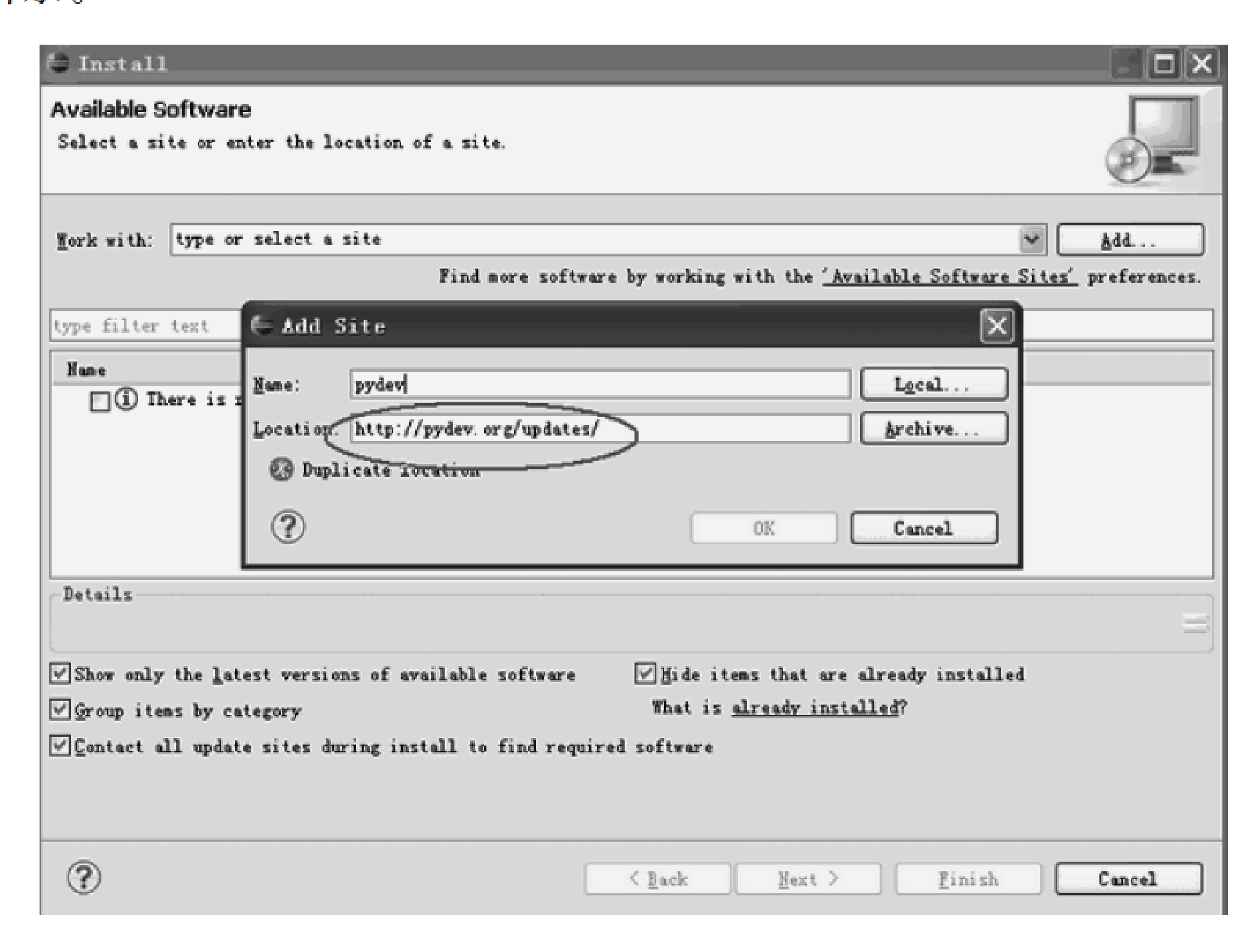



图 1.13 添加安装地址

完成后单击 OK 按钮，接着单击 PyDev 前面的"+"，展开 PyDev 的结点。这里要等一小段时间，让它从网上获取 PyDev 的相关组件，当完成后会多出 PyDev 的相

关组件在子结点里，勾选它们，然后单击 Next 按钮进行安装，如图 1.14 所示。

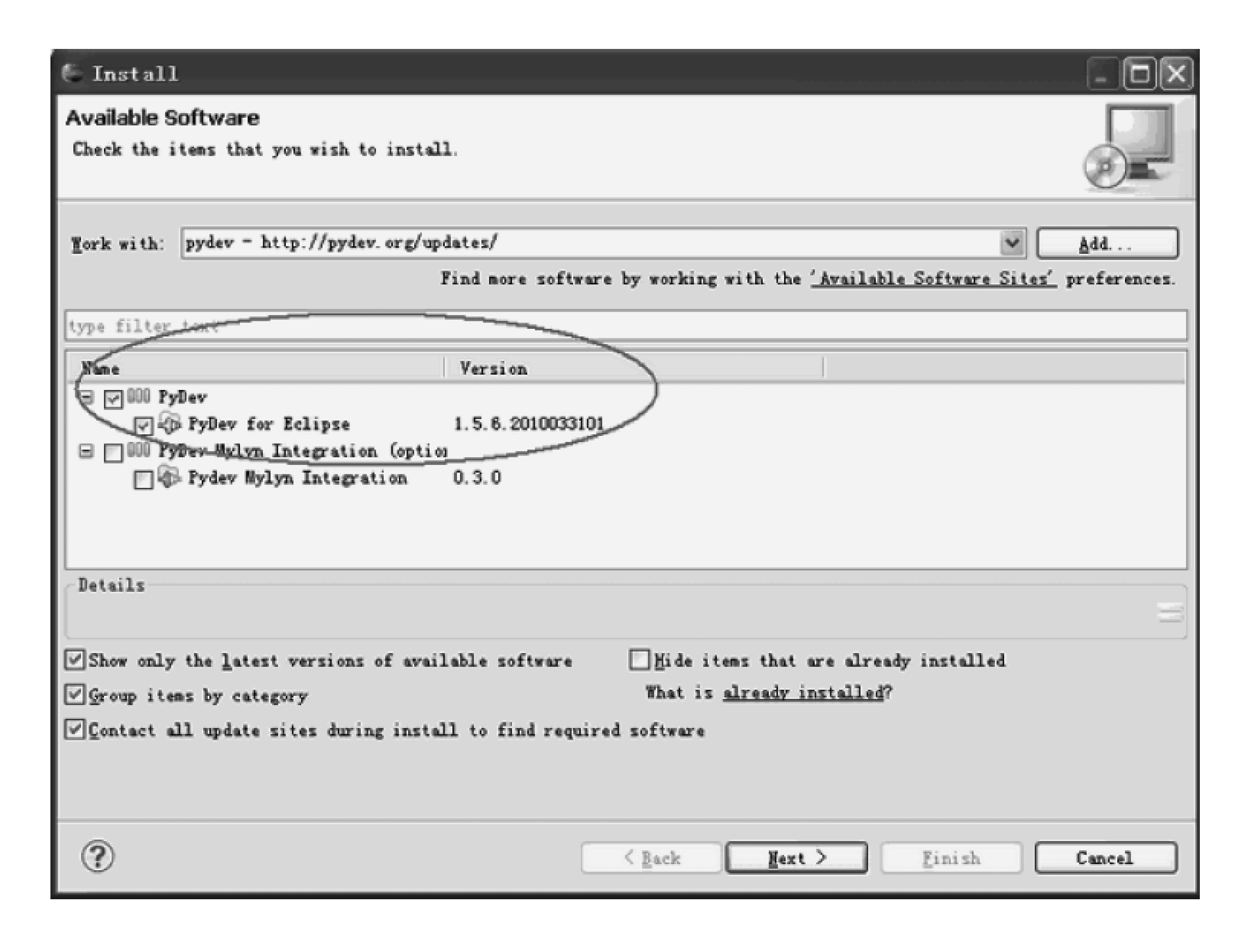



图 1.14　勾选相关组件

安装完成后重启 Eclipse 即可。

### 3. 设置 PyDev

在安装完成后还需要设置一下 PyDev，选择 Window→Preferences 命令打开 Preferences 对话框来设置 PyDev。首先设置 Python 的路径，在 PyDev 的 Interpreter-Python 页面中单击 New 按钮，如图 1.15 所示。

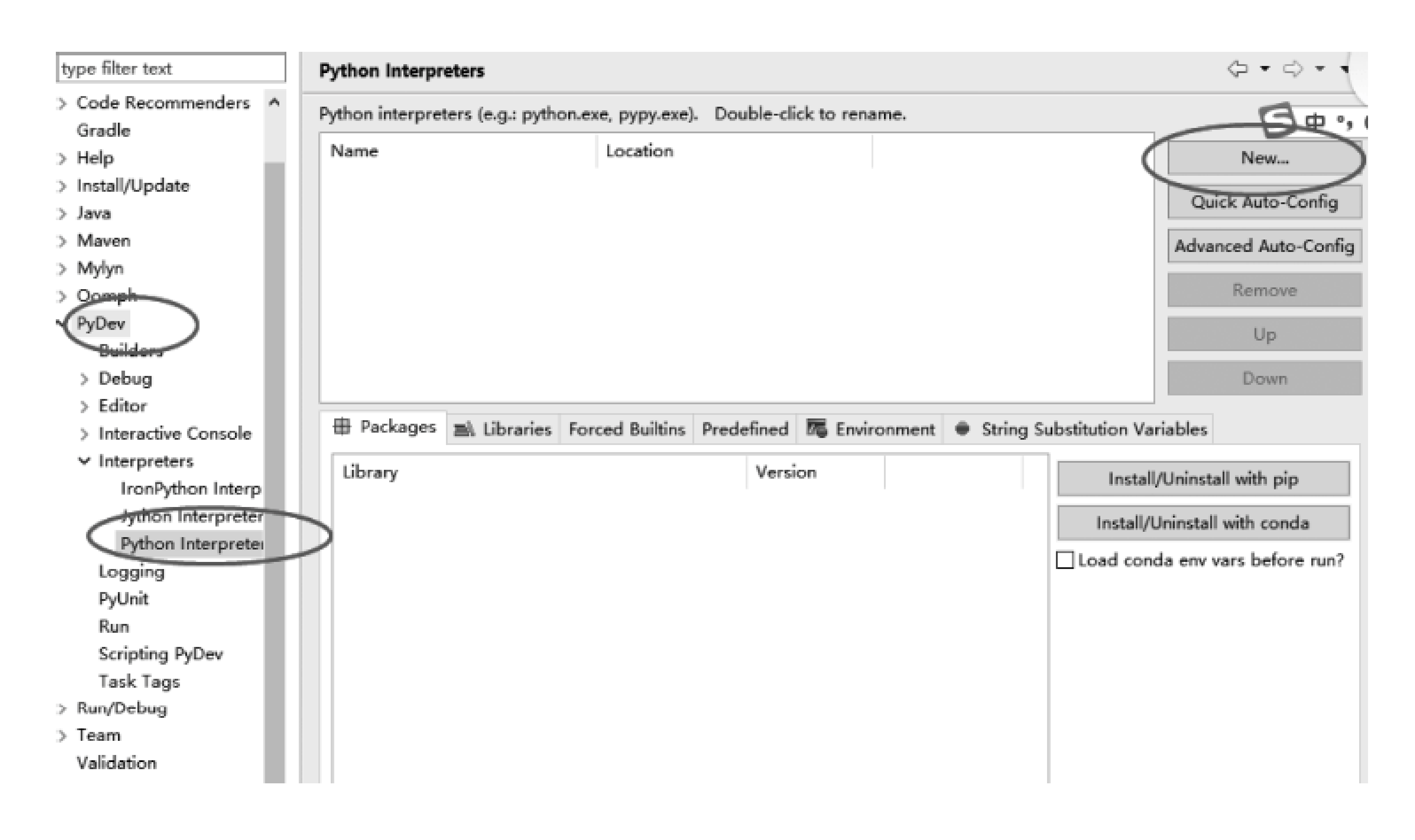



图 1.15　PyDev 的设置

此时会弹出一个对话框让用户选择 Python 的安装位置，选择自己安装 Python 的所在位置，如图 1.16 所示。

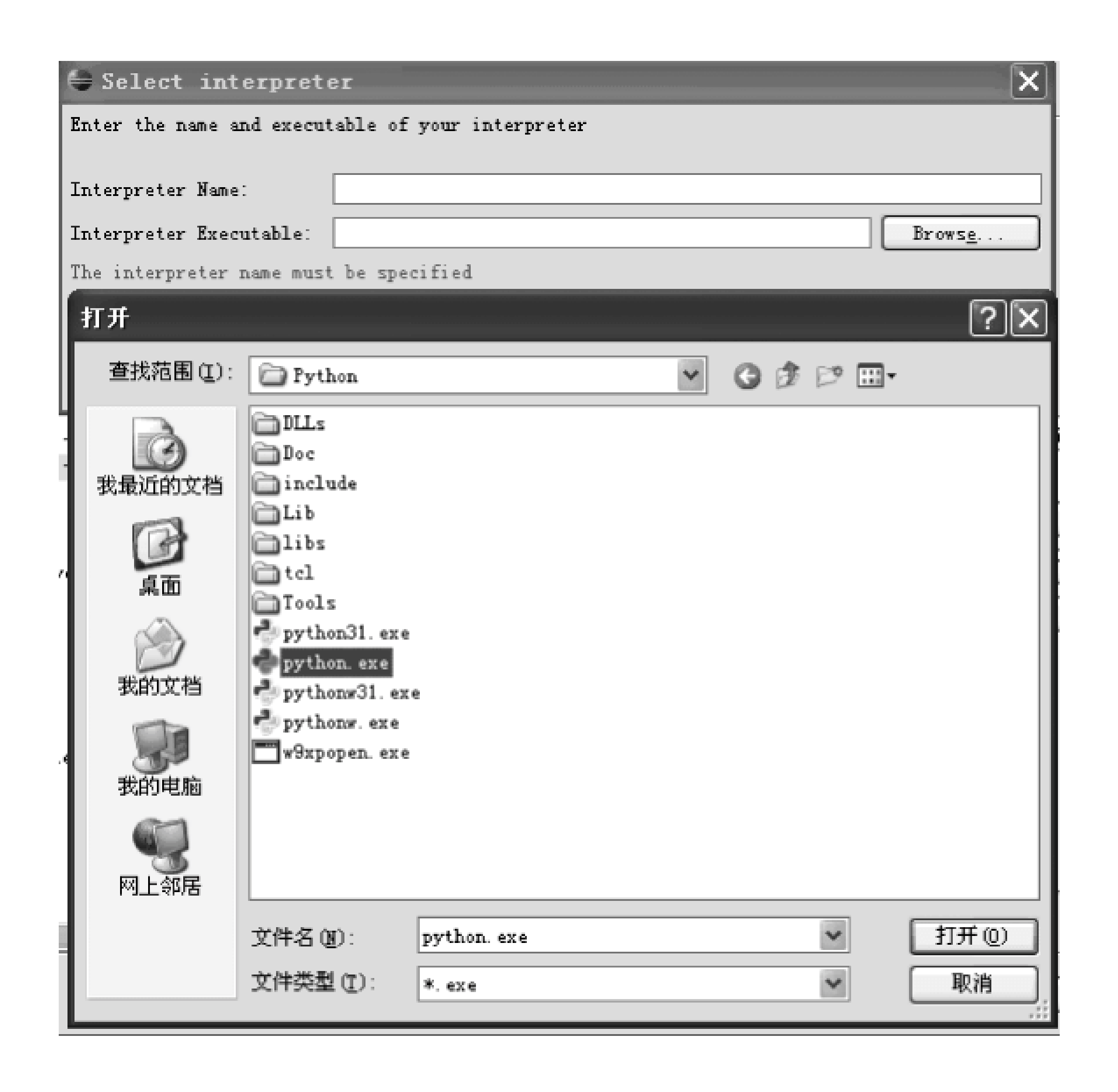



图 1.16　PyDev 的设置完成

完成之后 PyDev 就设置好了，用户可以开始使用。

#### 4. 建立 Python Project

在安装好 Eclipse＋PyDev 以后，用户就可以开始使用它来开发项目了。首先要创建一个项目，选择 File→New→Pydev Project 命令，如图 1.17 所示。

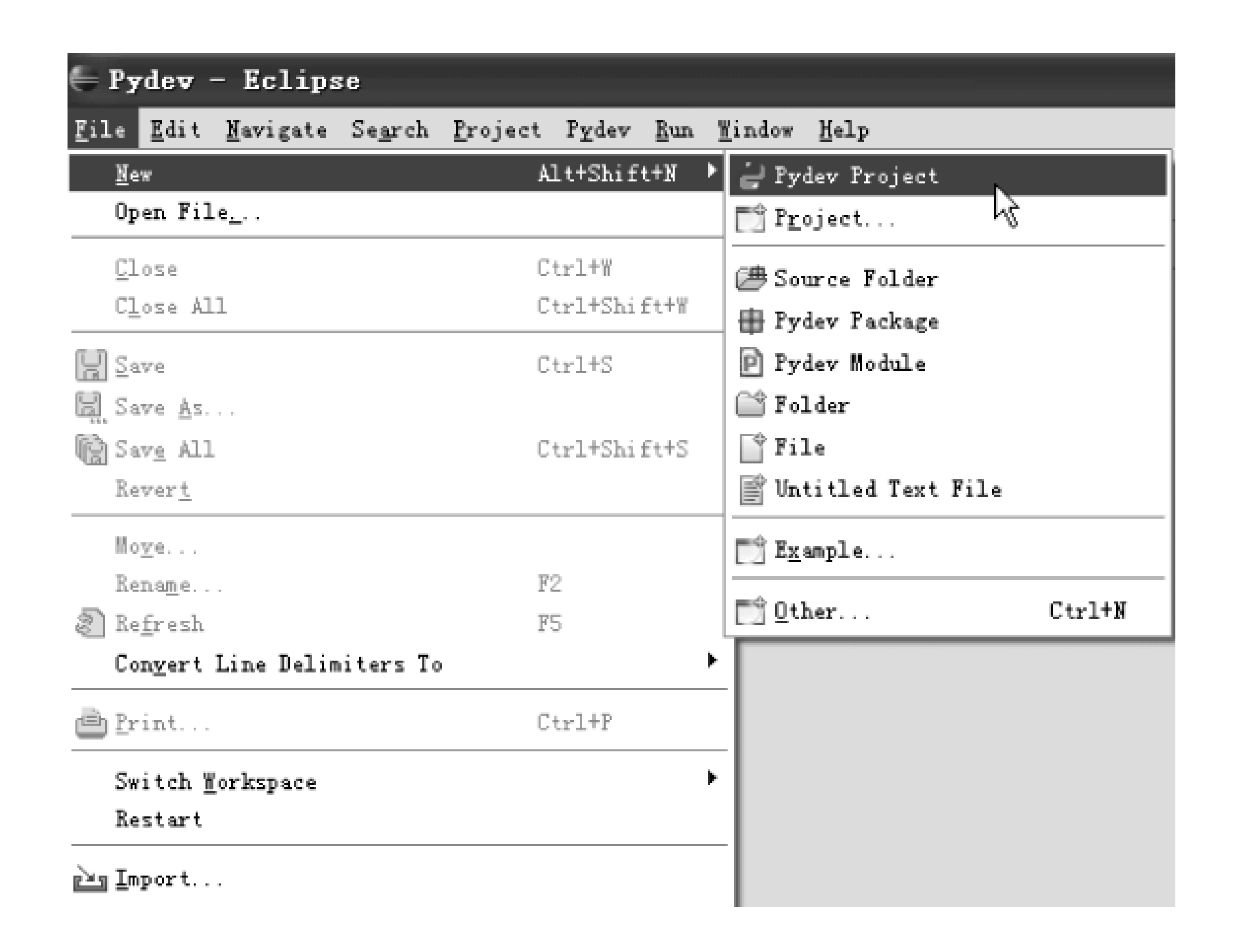



图 1.17　选择 Pydev Project 命令

此时会弹出一个对话框，填写项目名称以及项目保存地址，然后单击 Next 按钮完成项目的创建，如图 1.18 所示。

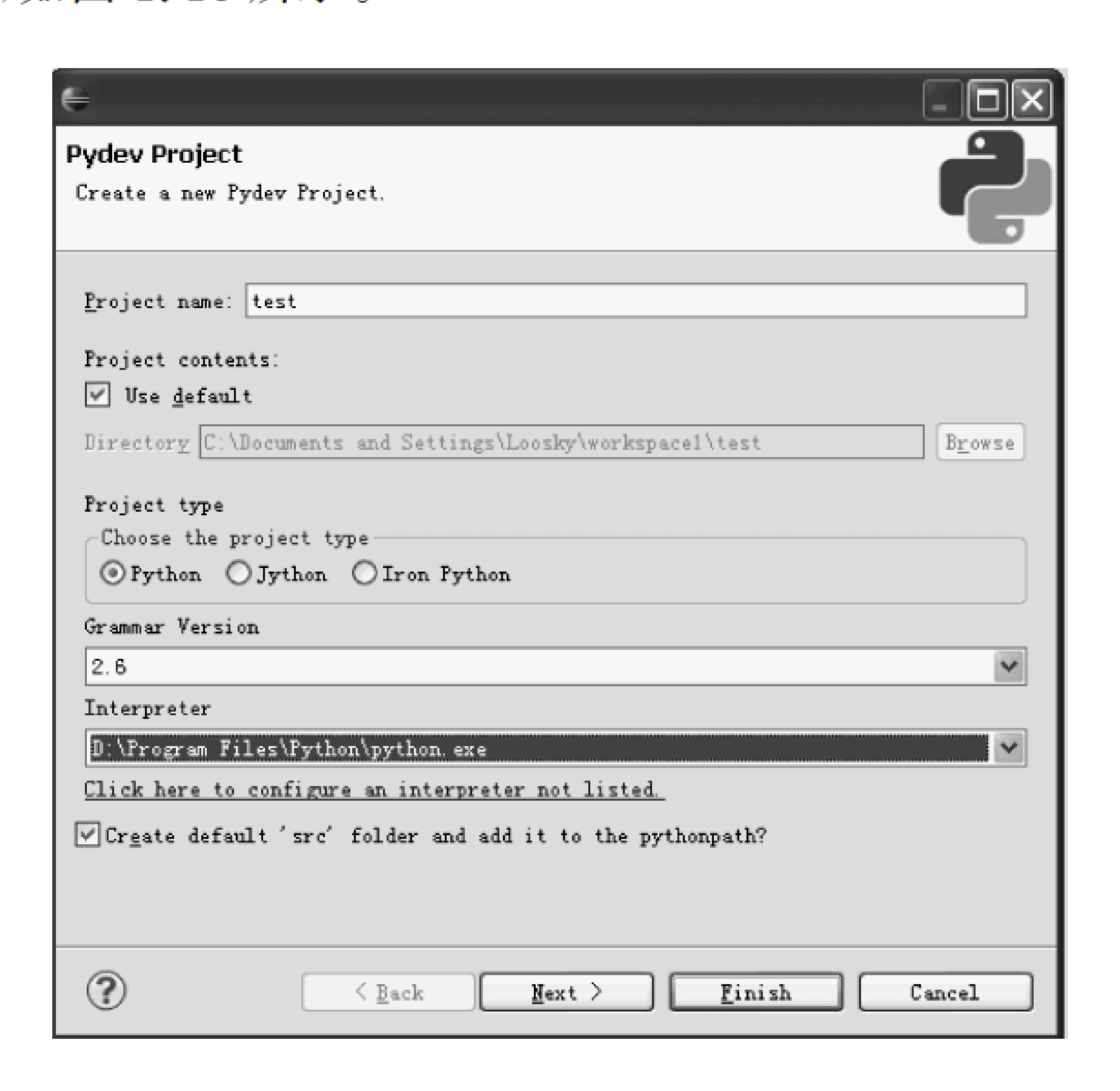



图 1.18 填写项目名称以及项目保存地址

### 5. 创建新的 PyDev Module

只有项目是无法执行的，接着必须创建新的 PyDev Module，选择 File→New→Pydev Module 命令，如图 1.19 所示。

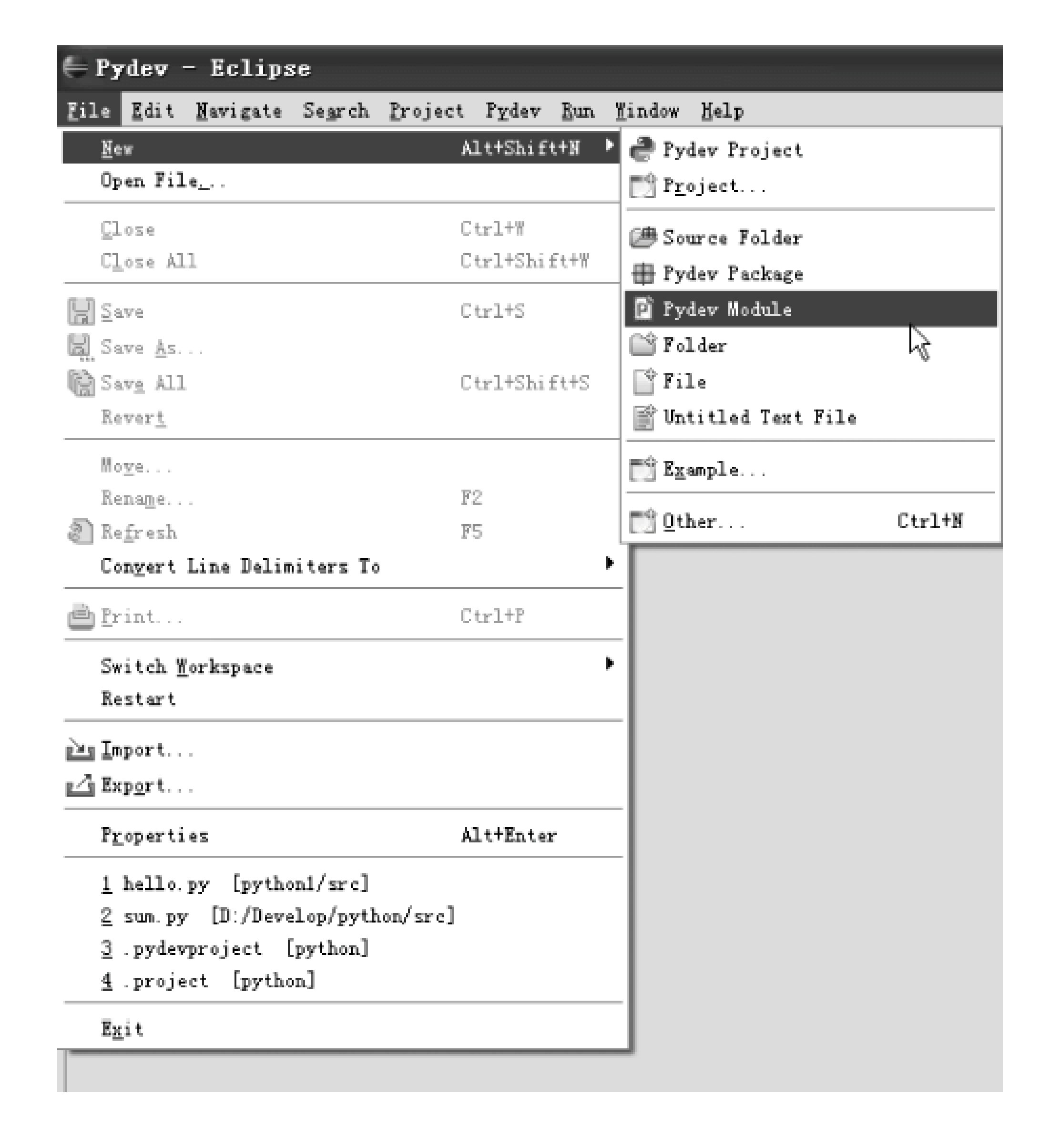



图 1.19 选择 Pydev Module 命令

在弹出的对话框中填写文件存放位置以及名称，注意名称不用加.py，系统会自动帮助用户添加。然后单击 Finish 按钮完成创建，如图 1.20 所示。

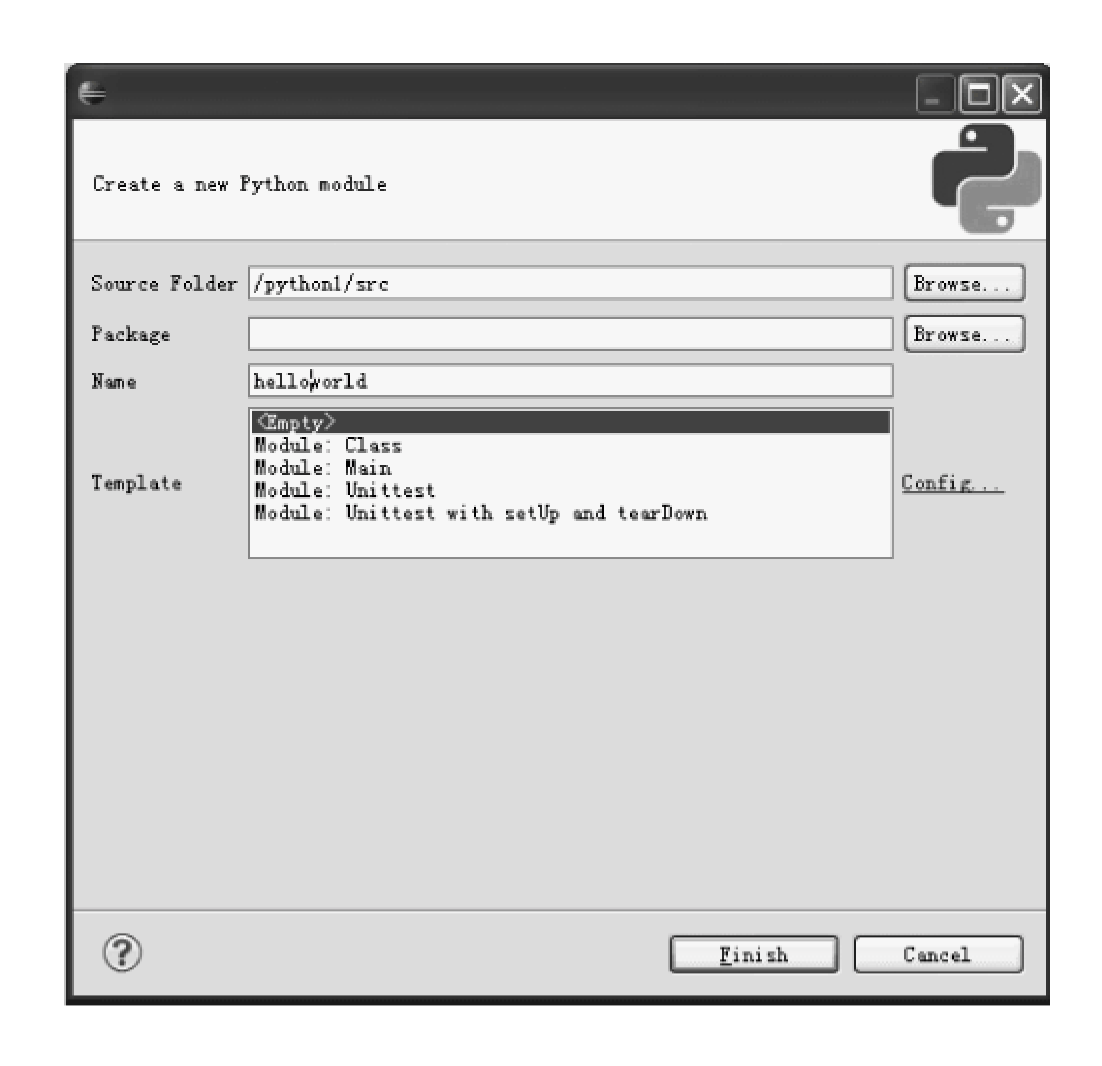



图 1.20　填写文件存放位置以及名称

输入代码"hello world!"，如图 1.21 所示。

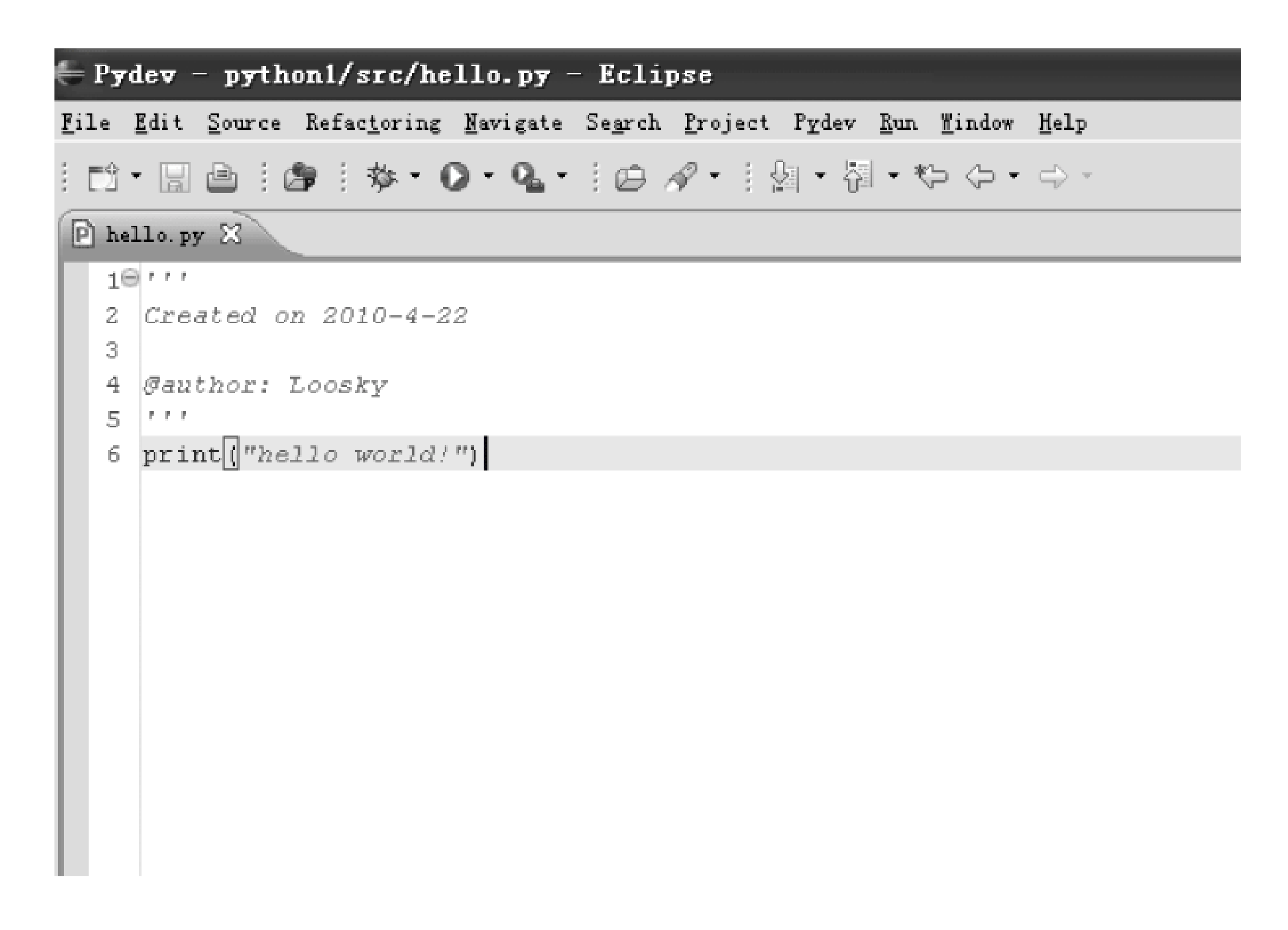



图 1.21　输入代码

### 6. 执行程序

程序写完后可以执行程序，在上方的工具栏中单击用于执行的按钮，如图 1.22 所示。

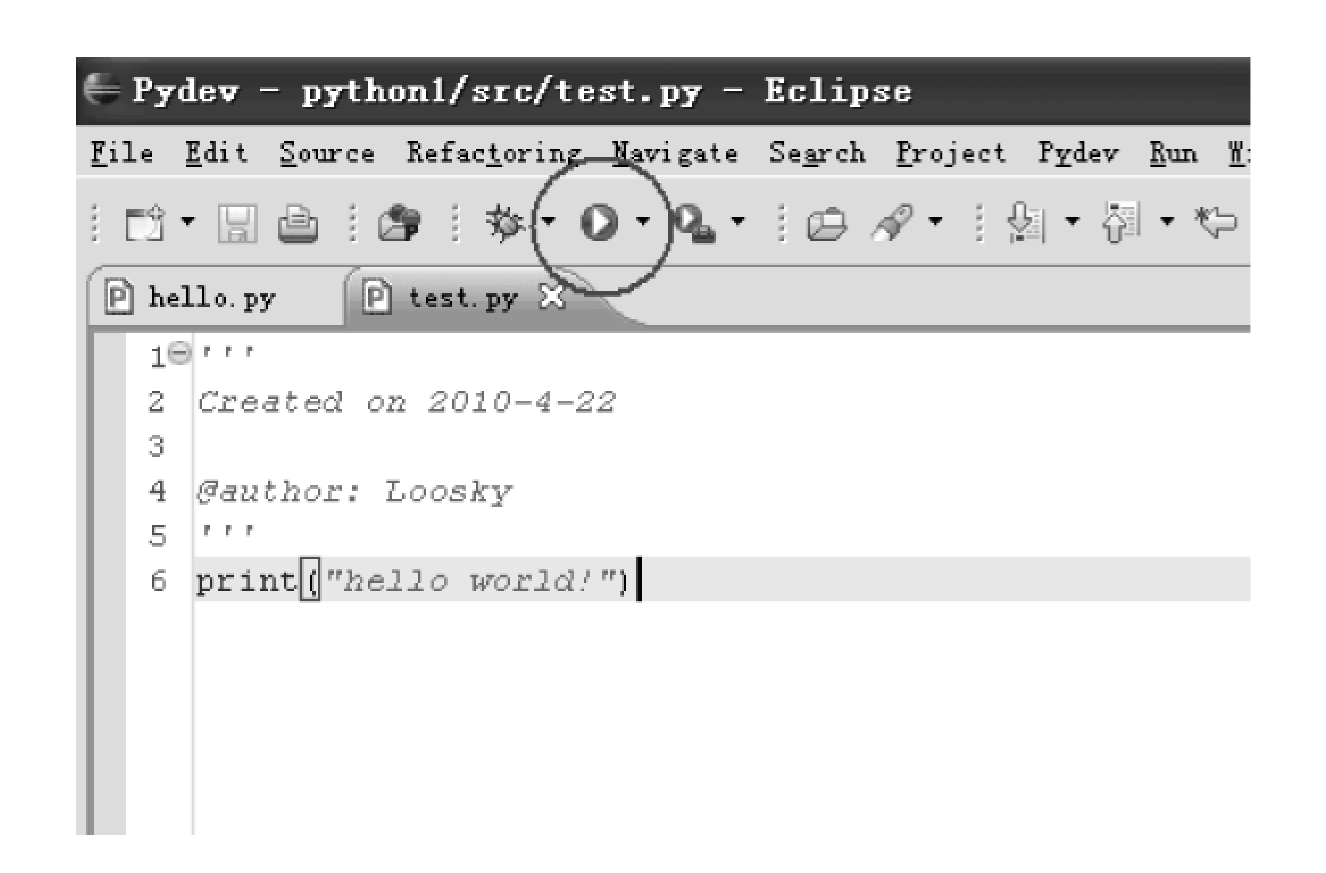



图 1.22 执行程序

此时会弹出一个让用户选择执行方式的对话框，通常选择 Python Run（见图 1.23），开始执行程序。

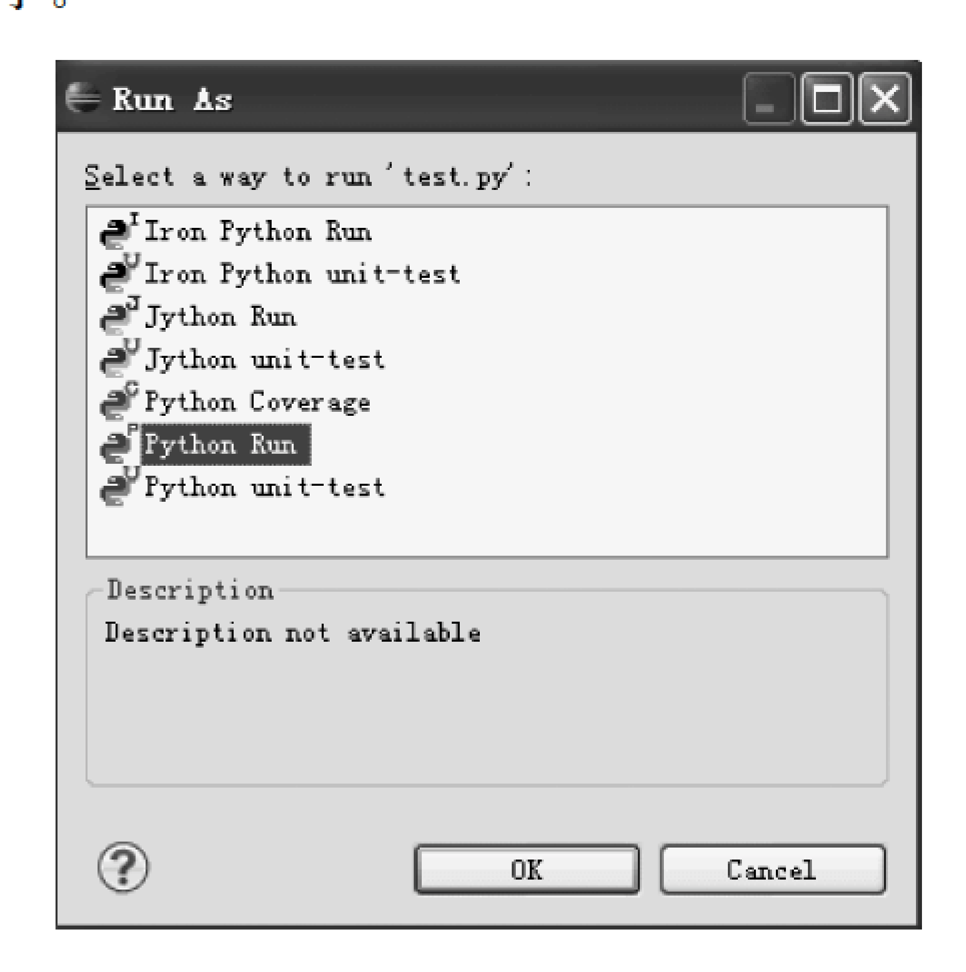



图 1.23 选择 Python Run

## 1.5 代码风格

下面介绍 Python 的代码风格。

### 1. 缩进

空格是最优先的缩进方式，每级缩进 4 个空格，连续行的折叠元素应该对齐。

```
#与起始定界符对齐
foo = long_function_name(var_one, var_two,
                         var_three, var_four)
#使用更多的缩进,以区别于其他代码
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)
#悬挂时应该增加一级缩进
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
#悬挂不一定要 4 个空格
foo = long_function_name(
  var_one, var_two,
  var_three, var_four)
```

值得注意的是,两个字符组成的关键字(例如 if),加上一个空格,加上开括号,为后面的多行条件创建了一个 4 个空格的缩进,这会给嵌入 if 内的缩进语句产生视觉冲突,因为它们也被缩进 4 个空格。

```
#不增加额外的缩进
if (this_is_one_thing and
    that_is_another_thing):
    do_something()
#添加一行注释,这将为支持语法高亮的编辑器提供一些区分
   if (this_is_one_thing and
        that_is_another_thing):
        #当两个条件都是真时将要执行
#在换行的条件语句前增加额外的缩进
```

多行结构中的结束花括号/中括号/圆括号应该是最后一行的第一个非空白字符,或者是最后一行的第一个字符。

```
my_list = [
    1, 2, 3,
    4, 5, 6,
]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
)
```

### 2. 制表符

Python 3 不允许混合使用制表符和空格来缩进。在 Python 2 的代码中若混合使用制表符和空格的缩进，应该转化为完全使用空格。在调用 Python 命令行解释器时若使用-t 选项，可以对代码中不合法的混合制表符和空格发出警告(warnings)；使用-tt 时警告(warnings)将变成错误(errors)，这些选项是被高度推荐的。

### 3. 最大行长度

Python 采取保守做法，要求行限制到 79 个字符(文档字符串/注释到 72 个字符)。折叠长行的首选方法是使用 Python 支持的圆括号、方括号(brackets)和花括号(braces)内的行延续。长行可以在表达式外面使用小括号来变成多行。连续行使用反斜杠更好。例如，长的多行的 with 语句不能用隐式续行，可以用反斜杠：

```
with open('/path/to/some/file/you/want/to/read') as file_1:\
    open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

### 4. 换行应该在二元操作符前面还是后面

几十年来推荐的风格是在二元操作符后面换行，但这会在两方面影响可读性——操作符往往分散在屏幕的不同列上，并且每个操作符都远离其操作数。

```
# 坏的：操作符远离它们的操作数
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

为了解决这个可读性问题，数学家和出版商遵循相反的约定。Donald Knuth 在他的“计算和排版”系列中解释了传统规则：“虽然一段内的公式总是在二进制操作和关系之后换行，但在二进制操作之前，显示的公式总是会换行”。

在 Python 代码中允许打破之前或之后一个二元运算符的规则，只要与当前惯例是一致的即可。建议使用 Knuth 的风格。

```
#好的：易于匹配操作数和操作符
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)

#二元运算符首选的换行位置在操作符后面
class Rectangle(Blob):
    def __init__(self, width, height,
                 color = 'black', emphasis = None, highlight = 0):
        if width == 0 and height == 0 and
                color == 'red' and emphasis == 'strong' or
                highlight > 100):
            raise ValueError("sorry, you lose")
        if width == 0 and height == 0 and (color == 'red' or
                                            emphasis is None):
            raise ValueError("I don't think so -- values are %s, %s" %
                             (width, height))
        Blob.__init__(self, width, height,
                      color, emphasis, highlight)
```

### 5. 空行

通常用两个空行分割顶层函数和类的定义，类内方法的定义用单个空行分割，额外的空行可以被用于分割一组相关函数(groups of related functions)，在一组相关的单句中间可以省略空行(例如一组哑元(a set of dummy implementations))。Python 接受换页符作为空格，Emacs(和一些打印工具)视这个字符为页面分割符，因此在文件中可以用它们为相关片段(sections)分页。

### 6. 源文件编码

Python 发行版本中的核心代码应该始终使用 UTF-8(Python 2 中使用 ASCII)。使用 ASCII(Python 2 中)或 UTF-8(Python 3 中)的文件不应该有编码声明。在标准库中，非默认编码仅用于测试目的，否则使用\x、\u、\U 或\N 是将非 ASCII 数据包含在字符串中的首选方式。

Python 3 及以上版本为标准库规定了以下策略：Python 标准库中的所有标识符必须使用 ASCII 标识符，并且尽可能使用英文单词。

另外，字符串和注释必须使用 ASCII，唯一的例外是测试非 ASCII 功能的用例和

作者名,名称不是基于拉丁字母表的作者必须提供这个字符集中他们名字的音译。开源项目面向全球,鼓励采用统一策略。

### 7. 导入

import 文件通常被放置在文件的首部,紧跟在模块注释和文档字符串之后,以及模块的全局变量和常量之前。

import 文件应该有顺序地导入以下库:

(1) 标准库。

(2) 相关的第三方库。

(3) 本地库。

用户应该在每组导入之间放置一个空行。

通常把任何相关规范放在导入之后,推荐使用绝对导入,因为它们更易读,并且如果导入系统的配置不正确(例如包中的一个目录在 sys. path 的最后),它们有更好的表现(或者至少给出更好的错误信息):

```
import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example
```

明确的相对导入可以被用来代替绝对导入,特别是在处理复杂的包布局时,绝对导入会产生不必要的冗余:

```
from . import sibling
from .sibling import example
```

标准库代码应该避免复杂包布局并使用绝对导入。隐式的相对导入永远不应该被使用,并且在 Python 3 中已经移除。

在从一个包含类的模块中导入类时,下面通常是好的写法:

```
from myclass import MyClass
from foo.bar.yourclass import YourClass
#如果这种写法导致本地名字冲突,那么就这样写
import myclass
import foo.bar.yourclass
#并使用"myclass.MyClass"和"foo.bar.yourclass.YourClass"来访问
```

避免使用通配符导入(from <模块名> import * ),因为这样不清楚哪些名字出现在命名空间,这会让读者和许多自动化工具"困惑"。

## 8. 字符串引号

Python 中单引号字符串和双引号字符串是一样的,当一个字符串包含单引号字符或双引号字符时,使用另一种字符串引号来避免字符串中使用反斜杠,这可以提高程序的可读性。

对于三引号字符串,Python 三引号允许一个字符串跨行,字符串中可以包含换行符、制表符以及其他特殊字符。

## 9. 表达式和语句中的空格

在以下情况应避免使用多余的空格:避免尾随空格,因为它通常是无形的,可能会让人感到很困惑,例如一个反斜杠后跟一个空格,换行符不会被视为行延续标记。有些编辑器不保留它,并且许多项目(例如 CPython 本身)都预先处理拒绝它的钩子。

始终在这些二元运算符两边放置一个空格:赋值(=)、比较(==、<、>、!=、<>、<=、>=、in、not in、is、is not),布尔运算(and、or、not)。

如果使用了不同优先级的操作符,在低优先级操作符周围增加空格(一个或多个)。注意不要使用多个空格,在二元运算符两侧添加的空格数量应相等。

```
i = i + 1
submitted += 1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
#不要在用于指定关键字参数或默认参数值的"="号周围使用空格
def complex(real, imag = 0.0):
    return magic(r = real, i = imag)
#函数注释应该对冒号使用正常规则,并且如果存在"->",两边应该有空格
def munge(input: AnyStr): ...
def munge() -> AnyStr: ...
#当结合一个参数注释和默认值时,在"="符号两边使用空格(仅仅当那些参数同时有注释和一个默认值时)
def munge(sep: AnyStr = None): ...
def munge(input: AnyStr, sep: AnyStr = None, limit = 1000): ...
#不要将多条语句写在同一行上
if foo == 'blah':
    do_blah_thing()
```

```
do_one()
do_two()
do_three()
#尽管有时可以将 if/for/while 和一小段主体代码放在一行,但在一个有多条子句的语句中不
#要如此,避免折叠长行
#最好不要
if foo == 'blah': do_blah_thing()
for x in lst: total += x
while t < 10: t = delay()
```

**10. 注释**

注释应该是完整的句子,如果注释是一个短语或句子,首字母应该大写,除非它是一个以小写字母开头的标识符(永远不要修改标识符的大小写)。

注释块通常应用在代码前,并和这些代码有着同样的缩进。注释块中的每行以“#”和一个空格开始(除非它是注释内的缩进文本)。注释块内的段落用仅包含单个“#”的行分割。

行内注释是和语句在同一行的注释,行内注释应该至少用两个空格和语句分开。它们应该以“#”和单个空格开始。如果行内注释所述是显而易见的,那么它就是不必要的。

```
#不要这样做
x = x + 1                    #增加 x
#但有时这样是有用的
x = x + 1                    #补偿 border
```

## 1.6 使用帮助

**1. Python Manuals**

自带 CHM 格式的 Python Manuals 存放在“\Python < x. x >\Doc\”目录下。

用户可以在 IDLE 界面下按 F1 键或单击 Help 选项下的 Python Docs 标签打开;也可以通过单击“开始”→Python x. x →Python Manuals 打开。

### 2. Module Docs

其中包含了 Python 中所有内置的和已经安装的第三方 Modules 文档信息。

单击“开始”→Python x. x→Module Docs，出现 pydoc 程序界面，用户可以在搜索框中直接查找需要的内容。

用户也可以单击 Open Browser 建立本地临时的 Web Server，浏览网页版的文档信息。当需要关闭时，单击“Quit Serving”即可。

利用 pydoc 手工在指定端口打开 Web Documentation Server，代码为"python -m pydoc -p 6789"（表示打开 pydoc 模块来查看 Python 文档，并在 6789 端口上启动 Web Server）。

然后访问“http://localhost：6789/”，可以看到所有已经安装的 Modules 文档信息。当需要关闭时，按 Ctrl+C 键终止命令或者关闭命令界面即可。

利用 pydoc 在终端下查看文档信息，在命令行中执行“python-m pydoc 函数名或模块名”可以看到自动生成的文档信息，按 Q 键退出。

例如查看 os 模块文档，代码为"python-m pydoc os"；在当前目录生成 dir()函数的 HTML 文档，代码为"python -m pydoc -w dir"。

在 Linux 下同样适合用 pydoc 方式查看文档。

### 3. help()和 dir()

1）查看对象信息：help([object])

（1）查看所有的 keyword：help("keywords")。

（2）查看所有的 modules：help("modules")。

（3）查看常见的 topics：help("topics")。

（4）查看模块：help("sys")。

（5）查看数据类型：help("list")。

（6）查看数据类型的成员方法：help("list. append")。

对于已定义或引入的对象，help([object])查询不使用单引号和双引号；对于未引入的模块等对象，要使用单引号或双引号。

2）查看当前属性及方法：dir()

查看对象的属性及方法用 dir([object])，dir([object])查询不使用单引号和双引号。

注意，help()函数多用来查看对象的详细说明，按Q键退出；dir()函数多用来查看对象的属性及方法，输出的是列表。

在使用help()和dir()之前要确定查询的对象已被定义或引入，否则会报错，例如“NameError：name is not defined”。

#### 4. 官方在线文档

其网址为“www.python.org/doc”。

#### 5. Q&A

- SegmentFault；
- Stack Overflow。

#### 6. 搜索与请教

Google、百度、必应、牛人等。

## 本章小结

本章主要介绍了Python语言发展的历史及其特点，然后分析了Python 2与Python 3不同的编程特点，最后以Eclipse with PyDev为例对Python的环境搭建进行了展示，接着对Python自带的IDLE界面和Python的各种开发环境进行了讲解。读者需要熟练掌握Python的编程特点，根据自身情况选择所需要的IDLE，并熟练地掌握相对应的Python开发工具。

## 习题

1. 在自己的计算机上建立一个Python入门使用的IDLE。在这个IDLE里，创建一个名为fruit的Python文件，并且打开它。

2. 在习题1建立完成的Python文件中，编写自己的第一个print(“Hello Python”)程序，并且修改文件名为HelloPython。

3. 简要阐述Python 2与Python 3的区别。

# 第 2 章



# Python语言基础知识

**本章学习目标：**

- 理解变量标识符的含义，掌握其构造和使用
- 熟练掌握 Python 中数据类型的基本运算
- 熟练掌握 Python 中条件语句的运用

本章先向读者介绍 Python 中最基本的标识符与变量的定义，并通过示例向读者展现 Python 语言的简单编写，详细地介绍数据类型的应用；接着深入讲解条件语句的类别、各自的作用以及循环语句的运用，最后通过实例展示 Python 在实际生活中的应用，简单地介绍 Python 中常用的函数。

## 2.1 标识符与变量

### 2.1.1 标识符

标识符(identifier)是用来标识某个实体的一个符号，在不同的应用环境下有不同的含义。

在日常生活中，标识符用来指定某个东西、人，要用到它、他或她的名字；在数学

中解方程时也经常用到这样或那样的变量名或函数名；在编程语言中，标识符是用户编程时使用的名字，对于变量、常量、函数、语句块也有名字。

在 Python 里，标识符由字母、数字、下画线组成，所以标识符可以包括英文、数字以及下画线(_)，但不能以数字开头。Python 中的标识符是区分大小写的。

另外，以下画线开头的标识符是有特殊意义的：以单下画线开头的(_foo)代表不能直接访问的类属性，需通过类提供的接口进行访问，不能用“from xxx import *”导入；以双下画线开头的(__foo)代表类的私有成员；以双下画线开头和结尾的(__foo__)代表 Python 里特殊方法专用的标识，例如__init__()代表类的构造函数。

### 2.1.2 变量

变量来源于数学，在计算机语言中能储存计算结果或能表示值的抽象概念。变量的表现形式是以一个绑定该变量的语句开始的，换句话说，变量可以通过变量名访问。在 Python 中没有变量声明，下面通过示例来具体看一下 Python 中变量的应用。

#### 1. 运行 hello world. py 时发生的情况

在运行 hello world. py 时，Python 都做了些什么呢？下面来深入研究一下。实际上，即便是运行简单的程序，Python 所做的工作也相当多。例如：

```
print("hello world!")
```

运行上述代码，用户将看到如下输出：

```
hello world!
```

在运行 hello world. py 文件时，末尾的. py 指出这是一个 Python 程序，因此 IDE 将使用 Python 解释器来运行它。Python 解释器读取整个程序，确定其中每个单词的含义。例如，当看到单词 print 时，解释器就会将括号中的内容打印到屏幕，而不管括号中的内容是什么。

在编写程序时，编辑器会以各种方式突出显示程序的不同部分。例如，编辑器知道 print 是一个函数的名称，因此将其显示为紫色；知道“hello world!”不是 Python 代码，因此将其显示为绿色。这种功能称为语法突出，在用户刚开始编写程序时很有帮助。

**2. 使用变量**

下面来尝试在 hello world. py 中使用一个变量。在这个文件开头添加一行代码,并对第 2 行代码进行修改:

```
message = "hello world! "
print(message)
```

运行这个程序,看看结果如何? 用户会发现,输出与以前相同:

```
hello world!
```

在这里添加了一个名为 message 的变量。每个变量都存储了一个值,即与变量相关联的信息,所存储的值为文本"hello world!"。添加变量导致 Python 解释器需要做更多的工作,在处理第 1 行代码时,它将文本"hello world!"与变量 message 关联起来;而处理第 2 行代码时,它将与变量 message 关联的值打印到屏幕。

下面进一步扩展这个程序:修改 hello world. py,使其再打印一条消息。为此,在 hello world. py 中添加一个空行,再添加下面两行代码:

```
message = "hello world! "
print(message)

message = "hello Python world! "
print(message)
```

现在如果运行这个程序,将看到两行输出:

```
hello world!
hello Python world!
```

在程序中可随时修改变量的值,而 Python 将始终记录变量的最新值。

**3. 变量的命名**

在 Python 中使用变量时需要遵守一些规则,违反这些规则将引发错误,请读者务必牢记下列有关变量的规则。

(1) 变量名只能包含字母、数字和下画线。变量名可以以字母或下画线打头,但

不能以数字打头，例如可以将变量命名为 name_1，但不能将其命名为 1_name。

(2) 变量名不能包含空格，但可以使用下画线来分隔其中的单词。例如，变量名 greeting_message 可行，但变量名 greeting message 会引发错误。

(3) 不要将 Python 关键字和函数名用作变量名，即不要使用 Python 保留的用于特殊用途的单词，例如 print。

(4) 变量名应既简短又具有描述性。例如，name 比 n 好，student_name 比 s_n 好，name_length 比 length_of_persons_name 好。另外，慎用小写字母 l 和大写字母 O，因为它们可能被人错看成数字 1 和 0。

如果想创建良好的变量名，需要经过一定的实践，在程序复杂时尤其如此。随着编写的程序越来越多，并参考阅读别人编写的代码，用户将越来越善于创建有意义的变量名。注意，就目前而言，应使用小写的 Python 变量名。在变量名中使用大写字母虽然不会导致错误，但避免使用大写字母是个不错的选择。

### 4. 在使用变量时避免命名错误

程序员都会犯错，而且大多数程序员每天都会犯错。优秀的程序员也会犯错，但他们知道如何高效地消除错误。下面来看一种读者可能会犯的错误，并学习如何消除它。这里将有意地编写一些引发错误的代码。请输入下面的代码，包括其中拼写不正确的单词 mesage：

```
message = "hello world!"
print(mesage)
```

当程序存在错误时，Python 解释器将竭尽所能地帮助用户找出问题所在。当程序无法成功运行时，解释器会提供一个 traceback。traceback 是一条记录，指出了解释器尝试运行代码时在什么地方陷入了困境。下面是用户不小心错误地拼写了变量名时 Python 解释器提供的 traceback。

```
Traceback <most recent call last>:
  File "<stdin>", line 1, in <module>
NameError: name 'mesage' is not defined
```

解释器指出它发现的是什么样的错误(见第 3 行)。在这里解释器发现了一个名称错误，并指出打印的变量 mesage 未定义，Python 无法识别用户提供的变量名。名

称错误通常意味着两种情况，即使用变量前忘记给它赋值，以及输入变量名时拼写不正确。

在这个示例中，第 2 行的变量名 mesage 中遗漏了字母 s。Python 解释器不会对代码做拼写检查，但要求变量名的拼写一致。例如，如果在代码的另一个地方也将 message 错误地拼写成了 mesage，结果将如何呢？

```
mesage = "hello world! "
print(mesage)
```

在这种情况下，程序将成功运行：

```
mesage = "hello world! "
print(mesage)
hello world!
```

计算机虽然"一丝不苟"，但不关心拼写是否正确。因此，在创建变量名和编写代码时用户无须考虑英语中的拼写和语法规则。其实，很多编程错误都很简单，只是在程序的某一行输错了一个字符。

## 2.2 数据类型及运算

Python 程序的运行取决于该程序要处理的数据。Python 中的所有数据值都是对象，并且每个对象或者值都有一个类型(type)。对象的类型确定了该对象支持哪些操作，也就是确定了可以对该数据值执行哪些操作。另外，类型还确定了该对象的属性(attribute)和项目(item)，以及该对象是否可以被改变。一个可以被改变的对象称为"可变对象"(mutable object)，而不可以被改变的对象称为"不可变对象"(immutable object)。内置的 type(obj)可以接受任何对象作为参数，并返回 obj 类型的对象。如果对象 obj 具有类型 type(或者其任何子类)，则内置函数 isinstance(obj, type)将返回 True，否则该函数将返回 False。

对于一些基本数据类型，比如数字、字符串、元组、列表和字典，Python 都有内置类型，用户可以使用 type()函数来确定 Python 给一个变量分配了什么数据类型。在下面的示例中，读者可以看到两个变量分配了不同的数据类型。

```
>>> coffee_cup = "coffee"
>>> type(coffee_cup)
<class 'str'>
>>> cups_consumed = 2
>>> type(cups_consumed)
<class 'int'>
```

Python 给 coffee_cup 分配了数据类型 str(字符串或字符串常量),因为它看到了引号括起来的一个字符串。然而,对于变量 cups_consumed,Python 看到了一个整数,因此给它分配了数据类型 int(整数)。

下面主要介绍数字类型数据,对于其他类型数据将在后面的章节中进行介绍。程序开发者还可以创建用户自定义类型,这些类型也被称为类(class)。

Python 支持 3 种不同的数值类型,即整数(int)、浮点数(float)、复数(complex)。

### 2.2.1 数据类型


视频讲解

Python 中的内置数字对象支持整数(普通整型和长整型)、浮点型数字和复数。Python 中的所有数字都是不可变对象,这意味着在对一个数字对象执行任何操作时总是会产生一个新的数字对象。

注意,数字常量不包含+或-,如果包含这两个符号,则表示该符号是一个分隔运算符。

#### 1. 整数型

在 Python 中整数常量可以是十进制、八进制或十六进制。十进制常量可以由第一个数字为非零数字的数字序列表示。为了表示八进制常量,可以在 0 后面带一个八进制数字(0～7)序列。为了表示十六进制常量,可以在 0x 后面带一个十六进制数字序列(0～9 和 A～F,可以使用大写或小写字母)。与 C 语言中所指的整型规则相同。

实际上,开发者不需要担心普通整型和长整型之间的区别,因为在需要的时候对普通整型的操作将生成长整型结果(也就是说,在运算结果超出普通整型结果的数值范围时)。不过,开发者可以选择将字母 L(或 l)放在任何类型的整数的后面,以明确地表示该整数是长整型。

长整型没有预定义的大小限制，只要内存允许，长整型可以无限大。另外，普通整型只占用了几字节的内存，并且其最小和最大值是由计算的架构决定的。sys. maxint 是可以使用的最大正整数，-sys. maxint-1 是可以使用的最大负整数。在 32 位计算机上，sys. maxint 是 2147483647。

**2. 浮点数**

Python 将带小数点的数字都称为浮点数。大多数编程语言都使用了这个术语，它指出这样一个事实：小数点可出现在数字的任何位置。每种编程语言都要细心设计，以妥善地处理浮点数，确保不管小数点出现在什么位置，数字的行为都是正常的。

从很大程度上说，在使用浮点数时都无须考虑其行为。用户只需输入要使用的数字，Python 通常都会按用户期望的方式处理它们。

**3. 复数**

复数是由两个浮点值组成的，一个是实部，一个是虚部，可以理解为数学中的无理数。开发者可以通过只读属性 z. real 和 z. imag 访问复数对象 z 的两个部分。

开发者可以将一个虚数字面常量指定为一个浮点或十进制字面常量，后面跟一个 j 或 J，例如：

0j、0. j、0. 0j、1j、1. j、1. 0j、1e0j、1. e0j

其末尾的 j 表示－1 的平方根。复数通常在电气工程中使用，在有些其他语言中使用 i 来表示，但 Python 选择使用 j。若想表示任何复数常数，可以使用一个浮点数(或整数)加上或减去一个虚数。例如要想表示数字 1 的复数，可以使用表达式 1＋0j 或者 1. 0＋0. 0j。

## 2.2.2 运算符和表达式


视频讲解

表达式(expression)是一个代码语句，Python 将计算这段表达式产生一个结果。在表 2.1 中列出了常见的运算符，其中运算符按优先级递减的顺序排列，高优先级在前，低优先级在后，列在一起的运算符具有相同的优先级。表 2.1 中的第 3 列给出了运算符的结合规则，即 L(从左到右)、R(从右到左)或 NA(无结合规则)。

表 2.1 运算符的结合规则

| 运 算 符 | 说 明 | 结合规则 |
|---|---|---|
| 'expr,…' | 字符串转换 | NA |
| {key: expr,…} | 创建字典 | NA |
| [expr,…] | 创建列表 | NA |
| (expr,…) | 创建元组或圆括号 | NA |
| f(expr,…) | 函数调用 | L |
| x[index: index] | 切片 | L |
| x[index] | 索引 | L |
| x.attr | 属性应用 | L |
| x ** y | 幂(x 的 y 次幂) | R |
| ~x | 按位非 | NA |
| +x、-x | 一元正和负 | NA |
| x * y、x/y、x//y、x%y | 乘除法、截断除法、求余 | L |
| x+y、x-y | 加减法 | L |
| x<<y,x>>y | 左移位、右移位 | L |
| x&y | 按位与 | L |
| x^y | 按位异或 | L |
| x\|y | 按位或 | L |
| x<y、x<=y、x>y、x>=y、x<>y、x!=y、x==y | 比较大小 | NA |
| x is y、x is not y | 同一性测试 | NA |
| x in y、x not in y | 成员测试 | NA |
| not x | 布尔"非" | NA |
| x and y | 布尔"与" | L |
| x or y | 布尔"或" | L |
| Lambda arg,…expr | 匿名简单函数 | NA |

**注意**:<>、!=是"不等于"运算符的两种表现形式。

在表 2.1 中,expr、key、f、index、x 和 y 表示任意表达式,而 attr 和 arg 表示任何标识符。除了字符串转换之外,符号"…"表示使用逗号分隔的零个或多个重复表达式,字符串转换需要一个或多个重复表达式。除了字符串转换运算符之外,在所有运算符中,拖尾逗号允许使用且没有危害,在字符串运算符中禁止使用拖尾逗号。

下面以在 Python 中对整数执行简单的加(+)、减(-)、乘(*)、除(/)运算为例进行介绍:

```
>>> 1 + 1
2
>>> 9 - 8
```

```
1
>>> 3 * 7
21
>>> 5/2
2.5
```

在终端会话中,Python 直接返回运算结果。

Python 使用两个乘号表示乘方运算:

```
>>> 2 ** 3
8

>>> 3 ** 3
27

>>> 10 ** 4
10000
```

Python 还支持运算次序,因此用户可以在同一个表达式中使用多种运算。用户还可以使用括号来修改运算次序,让 Python 按自己指定的次序执行运算,例如:

```
>>> 5 + 3 * 5
20

>>> (5 + 3) * 5
40
```

在这些示例中,空格不影响 Python 计算表达式的方式,它们的存在旨在让机器阅读代码时能迅速确定先执行哪些运算。

浮点型数据的运算与整数类似,例如:

```
>>> 0.1 + 0.3
0.4

>>> 2.2 * 0.5
1.1
>>> 0.6/0.4
1.4999999999998
```

需要注意的是,结果包含的小数位数可能是不确定的(例如 0.6/0.4)。所有语言

都存在这种问题，用户没有什么可担心的。Python 会尽力找到一种方式来尽可能精确地表示结果，但鉴于计算机内部表示数字的方式，这在有些情况下很难。就现在而言，暂时忽略多余的小数位数即可。

在 Python 中，当整型(int)与浮点型(float)做运算时，结果默认是浮点型，例如：

```
>>> 5/2.0
2.5
```

在 Python 中有 3 个布尔运算符，即 and、or 和 not。

布尔运算也叫逻辑运算，它的运算结果只有两个值，即真(True)和假(False)。一个布尔表达式也只有一个逻辑结果，要么为 True，要么为 False。例如 1<2 这样一个表达式，它的结果就是 True，而 3+1>5 这样一个表达式，它的结果就是 False。

and 也被称为“与”运算，该运算符连接左、右两个布尔表达式，即 x and y。若 x 和 y 都为 True，则 and 运算的结果为 True；否则，只要 x 和 y 中至少有一个为 False，整个 and 运算的结果就为 False。

or 也被称为“或”运算，该运算符连接左、右两个布尔表达式，即 x or y。若 x 和 y 都为 False，则 or 运算的结果为 False；否则，只要 x 和 y 中至少有一个为 True，整个 or 运算的结果就为 True。

not 也被称为“非”运算，该运算符作用于一个布尔表达式，即 not x。若 x 为 True，则 not 运算的结果为 False；若 x 为 False，则 not 运算的结果为 True。

例如：

```
>>> (5 > 3) and (3 + 4 == 7)          # 左、右两边都为 True, and 运算的结果为 True
True
>>> (5 < 3) and (3 + 4 == 7)          # 左边为 False, 右边为 True, and 运算的结果为 False
False
>>> (5 < 3) or (3 + 4 == 7)           # 左边为 False, 右边为 True, or 运算的结果为 True
True
>>> (5 < 3) or (4 + 2 == 7)           # 左、右两边都为 False, or 运算的结果为 False
False
>>> not (5 < 3)                       # 右边为 False, not 运算的结果为 True
True
```

布尔表达式在本章后面所介绍的分支结构控制语句和循环语句中特别有用，请读者务必掌握。

## 2.3 分支结构控制语句


视频讲解

说到分支结构控制语句，似乎所有的编程语言都有类似的语句。条件判断、有限循环、无限循环语句，这几个语句是最基本的，也是必不可少的。

### 2.3.1 if 语句

最基本的分支结构语句就是 if 语句了，在 Python 中 if 语句具有如下基本格式：

```
if condition: statement
```

如果用户曾经在其他编程语言中使用过 if 语句，这个格式可能看起来有些奇怪，因为语句中没有“then”这个关键字。其实在 Python 中“:”就起到了“then”的作用。机器会对括号内的条件进行求值判断，若返回 True 则执行冒号后的内容，若返回 False 则跳过冒号后的语句。

下面通过一些例子来展示如何使用 if 语句。假设有一个表示某人年龄的变量，若想知道这个人是否为能够驾驶汽车的年龄，可使用如下代码：

```
age = 19
if age >= 18:
    print("You are old enough to drive")
```

注意，在 print 前面要有空格，否则在运行过程中会出现错误。例如：

```
>>> age = 19
>>> if age >= 18:
    print("You are old enough to drive")
  File"<stdin>",line 2
    print("You are old enough to drive")
          ^
IndentationError: expected an indented block
```

在 if 语句中，缩进的作用与 for 循环中相同。如果测试通过了，将执行 if 语句后面所有缩进的代码行，否则将忽略它们。当语句正确无误时运行语句，可以看到在第 2 行 Python 检查变量 age 的值是否大于或等于 18。答案是肯定的，因此 Python 执行

第 3 行缩进的 print 语句，输出以下内容：

```
You are old enough to drive
```

在紧跟在 if 语句后面的代码块中，可根据需要包含任意数量的代码行。例如下面在一个人够驾驶汽车的年龄时再打印一行输出，问他是否拥有驾驶证：

```
>>> age = 19
>>> if age >= 18:
    print("You are old enough to drive!")
    print("Have you a drive license?")
```

条件测试通过了，而两条 print 语句都缩进了，因此它们都将执行：

```
You are old enough to drive!
Have you a drive license?
```

如果 age 的值小于 18，这个程序将不会有任何输出。

### 2.3.2 if-else 语句

在 if 语句中，如果条件返回 False，Python 将会移动到下一条语句中。如果希望在返回 False 时可以执行另一组语句，那么就需要用到条件语句了。似乎所有的条件语句都使用 if-else。它的作用可以简单地概括为“非此即彼”。当满足条件时执行 A 语句，否则执行 B 语句。下面看一下 Python 中 if-else 的具体应用：

```
>>> age = 20
>>> if age > 18:
    print("You are an adult")
else:
    print("You are underage")
...
You are an adult
```

在使用 if-else 语句的过程中需要注意如何放置 else 的位置，如果 else 缩进了，就会发生错误：

```
age = 20
if age > 18:
```

```
    print("You are an adult")
    else:
  File"<stdin>",line 3
    else:
        ^
SyntaxError: invalid syntax
>>>
```

### 2.3.3　if-elif-else 语句

到目前为止，大家学习了如何使用 if 或 if-else 控制语句，这使得用户对程序的控制具有更多的灵活性。其实这种语句还有很多。比如当需要将一个值与多个范围的条件进行比较时，使用 Python 提供的 if-elif-else 结构。Python 只执行 if-elif-else 结构中的一个代码块，它依次检查每个条件测试，在测试通过后，Python 将执行紧跟在它后面的代码，并跳过余下的测试。例如：

```
>>> age = 41
>>> if age < 18:
    print("You are underage.")
elif 18 <= age < 40:
    print("You are a young man.")
elif 40 <= age < 60:
    print("You are middle-aged.")
else:
    print("You are an old man.")
...
You are middle-aged.
```

Python 并不要求 if-elif 结构后面必须有 else 代码块。在有些情况下，else 代码块很有用；而在其他一些情况下，使用一条 elif 语句来处理特定的情形更清晰，例如：

```
>>> age = 20
>>> if age < 18:
        print("You are underage.")
    elif 18 <= age < 40:
        print("You are a young man.")
    elif 40 <= age < 60:
        print("You are middle-aged.")
    else:
        print("You are an old man.")
...
You are a young man.
```

else 是一条包罗万象的语句，只要不满足任何 if 或 elif 中的条件测试，其中的代码就会执行，这可能会引入无效甚至恶意的数据。如果用户知道最终要测试的条件，应考虑使用一个 elif 代码块来代替 else 代码块。这样，用户就可以肯定，仅当满足相应的条件时自己的代码才会执行。

## 2.4 循环语句

### 2.4.1 循环结构控制语句



视频讲解

在编程过程中，大家总会遇到需要把某个过程重复 N 次的情况，这个时候就要使用到循环语句来完成语句或函数等任务的重复执行。

#### 1. while 语句

在 Python 中，while 循环语句是“条件控制的”循环，即无限循环，只要不满足某种条件，任务就会一直执行，直到满足条件。例如：

```
>>> number = 1
>>> while number <= 10:
    print(number)
    number = number + 1
...
1
2
3
4
5
6
7
8
9
10
```

在这个示例中，while 语句会先检查测试条件变量 number 的值，只要该变量的值小于或等于 10，Python 语句就会执行。在第 4 行中，while 循环的最后一条语句会将变量的值加 1。因此，当 number 等于 11 时，while 的测试语句返回 False，这样一个循环就终止了。

**2. for 语句**

在 Python 中，for 循环结构称为"计数控制"循环，即有限循环。在循环任务中会出现需要执行一定次数的情况，这个时候可以使用 for 循环来完成这个任务。例如：

```
>>> numbers = [0,1,2,3,4,5,6,7,8,9]
>>> for number in numbers:
        print(number)
...
0
1
2
3
4
5
6
7
8
9
```

## 2.4.2　循环嵌套控制语句


视频讲解

循环嵌套是在一个循环语句中嵌入另一个循环语句。例如，位于循环代码块中的一个 for 循环就是循环嵌套。

## 2.4.3　break 语句和 continue 语句

通常，循环会不断地执行代码块，直到条件不满足时才会停止。但在有些情况下，用户可能想中断循环，开始进入"下一轮"代码块执行流程或直接结束循环。

要想结束或跳出循环，可以使用 break 语句。break 语句用于控制程序流程，可以使用它来控制哪些代码行将执行，哪些代码行不执行，从而让程序按用户的要求执行用户要执行的代码。例如从 1 到 10，但只打印其中与 3 乘积小于 10 的数：

```
>>> number = 0
>>> while number < 10:
```

```
    number += 1
    if number * 3 >= 10:
        break
    print(number)
```

首先将 number 设置成 0，由于它小于 10，Python 进入 while 循环。进入循环后，以步长为 1 的方式往上数(见第 3 行)，因此 number 为 1。接下来 if 语句检查 number 与 3 的求积运算结果，如果结果小于 10，就让 Python 执行余下的代码，将结果打印出来；如果结果大于或等于 10，就执行 break 语句，中断循环。

```
1
2
3
\\\
```

如果要返回到循环开头，并根据条件测试结果决定是否继续执行循环，可以使用 continue 语句，它不像 break 语句那样，不再执行余下的代码并退出整个循环。例如来看一个从 1 到 10，但只打印其中偶数的循环：

```
>>> number = 0
>>> while number < 10:
    number += 1
    if number % 2 == 0:
        continue
    print(number)
```

首先将 number 设置成了 0，由于它小于 10，Python 进入 while 循环。进入循环后，以步长为 1 的方式往上数(见第 3 行)，因此 number 为 1。接下来 if 语句检查 number 与 2 的求模运算结果，如果结果为 0(意味着 number 可以被 2 整除)，就执行 continue 语句，让 Python 忽略余下的代码，并返回到循环的开头；如果当前的数字不能被 2 整除，就执行循环中余下的代码，Python 将这个数字打印出来：

```
1
3
5
7
9
```

### 2.4.4　range()函数

Python 内置的 range()函数可以迭代地生成一组数字序列，这个功能在循环语句中特别有用，尤其是跟需要计数的 for 循环语句搭配使用可以大大降低代码量。例如打印 0～9 这 10 个数字，需要首先创建一个包含 0～9 这 10 个数字的数组，而通过 range()函数能够更加轻易地实现该功能：

```
>>> for number in range(10):
...         print(number)
...
0
1
2
3
4
5
6
7
8
9
```

如果 range()函数只有一个参数，那么参数表示数列的结束数字(小于该参数的最大数字)，且默认数字从 0 开始，例如输入参数 5，则生成从 0 开始直到小于 5 的最大数字(也就是 4)的数列，即“0 1 2 3 4”。如果为 range()函数传入两个参数，则第 1 个参数表示起始数字，第 2 个参数表示结束数字(小于该参数的最大数字)，例如 range(5，10)生成从 5 开始直到小于 10 的最大数字(也就是 9)的数列，即“5 6 7 8 9”。如果为 range()函数传入了第 3 个参数，则第 3 个参数表示步长，即每隔多少取一个数字，例如 range(0,10,3)表示从 0 开始直到 9 每隔 3 取一个数字组成的数列，即“0 3 6 9”。

len()函数用于获取一个数组的长度，如果将 range()函数与 len()函数相结合，那么就可以实现遍历数组的功能：

```
>>> arr = ['a', 'b', 'c', 'd', 'e']
>>> for i in range(len(arr)):
...     print(i,arr[i])
...
0  a
1  b
```

```
2  c
3  d
4  e
```

## 2.5 常见的 Python 函数

C、C++、Java、Ruby、Perl、Lisp 等众多编程语言的程序都是由函数和类组成的，当然 Python 程序中包含的不是函数就是类。

函数犹如小型程序，可用来执行特定的操作。Python 提供了很多函数，可用来完成很多神奇的任务。用户也可以自己编写函数(这将在后面更详细地介绍)。前面用的 print()就是一个函数。再比如，在学习运算符和表达式的时候曾用到 ** 表示乘方运算，实际上可以不使用这个运算符，而使用 pow()函数。

```
>>> 2 ** 3
8
>>> pow(2,3)
8
```

这里的 pow()就是 Python 中的一个标准函数，也称为内置函数。使用这种形式的函数称为调用函数：用户向它提供实参(即实际传给函数的参数，这里是 2 和 3)，而它返回一个值。鉴于函数调用返回一个值，因此它们也是一个表达式。

本节将会通过示例的方法介绍几种在 Python 中常用的函数。

### 1. print()函数

本节之前的代码实际上已经多次用到了 print()函数，这里专门对其用法进行介绍。print()函数主要用于打印输出，这在任何编程语言中都是最基本的功能。需要注意的是，print()只有在 Python 3 中才是一个函数，在 Python 2 中并不是函数，这就意味着在 Python 2 中 print 后面不需要括号。本书均以 Python 3 为例进行编写，因此读者会发现本书代码中 print 的后面总是带着括号，而其参数都写在括号中。

print()函数的第 1 个参数是期望打印的数据，它可以是字符串、数值、布尔、列表、字典等任何类型或其变量。下列代码显示了使用 print()函数输出各个类型数据的结果：

```
>>> x = 12
>>> print(x)   #打印数值变量
12
>>> s = 'python'
>>> print(s)   #打印字符串变量
python
>>> L = [1,2,3]
>>> print(L)   #打印列表变量
[1, 2, 3]
>>> t = ('a','b','c')
>>> print(t)   #打印元组变量
('a', 'b', 'c')
>>> b = True
>>> print(b)   #打印布尔变量
True
>>> d = {'a': 1, 'b':2}
>>> print(d)   #打印字典变量
{'b': 2, 'a': 1}
```

可以看到，无论是何种数据类型，print()函数都有办法以最合适的方式进行输出。关于列表、字典等数据结构的相关内容，请读者参考本书第 3 章。

在 print()函数中有两个重要的参数，即 sep 和 end。这两个参数可以让用户在输出时自定义间隔符和结束符。这里用下面的代码来说明这两个功能：

```
>>> print("hello", "world", "and", "python")
hello world and python
>>> print("hello", "world", "and", "python",sep = ", ")
hello, world, and, python
>>> print("hello", "world", "and", "python", end = ".")
hello world and python.>>>
```

从上述代码中可以看到，在不使用 sep 和 end 参数的情况下，当打印一连串字符串时，print()函数默认使用一个空格作为每个字符串之间的间隔符，并且在最后一个字符串的结尾使用了一个回车符作为结束符(虽然该回车符看不见，但是用户可以发现下面的 3 个右箭头符号“>>>”另起了一行)。上述代码的第 3 行修改了 sep 参数，将其设为逗号加空格，第 4 行显示了最终结果，可以看到每个字符串之间改为使用逗号加空格进行间隔。上述代码的第 5 行修改了 end 参数，将其设为句点符号，第 6 行显示了最终结果，可以看到打印的最后使用了句点作为结尾。尤其需要注意的是，由于将 end 参数默认的回车符设为了句点符，在结束时就没有换行，后面的 3 个箭头符号跟在了句点的右侧。

除了屏幕输出(也就是将数据结果显示在计算机屏幕上)以外,print()也支持文件输出(将数据输出到文件中进行保存)。为了保存文件,print()提供了一个名为 file 的参数,例如:

```
>>> out = open("test.txt", "w")
>>> print("Hello world", file = out)
```

上述代码首先打开了一个文件,名为 test.txt,然后使用 print()函数将字符串“Hello world”写入到了文件中。关于 Python 中文件的操作请参看本书第 7 章。

在打印输出大量字符串时,格式化功能特别重要,这让整个输出语句在格式上显得更为清晰,尤其是当需要将不同类型的变量拼接在一起的时候。Python 3 中的 print()函数提供了两种格式化的方式:一种方式兼容了 Python 2 格式化的语法,另一种方式采用了全新的 format()函数。下面先来看看传统的类 C 语言的格式化输出方式:

```
>>> str = "Hello world"
>>> length = len(str)
>>> print("字符串 %s 的长度是: %d" % (str, length))
字符串 Hello world 的长度是:11
```

上述代码的第 3 行中的%s 表示此处需要一个字符串进行填充,%d 表示此处需要一个带符号的整数进行填充,而后面的% (str, length)则表示用 str 和 length 这两个变量分别填充前面的%s 和%d,也就是将%s 所在的位置用 str 的值(也就是“Hello world”)进行替换,将%d 所在的位置用 length 的值(也就是 11)进行替换,替换以后 print()参数的字符串就变成了最后一行所看到的结果。除了%s 和%d 以外,Python 还提供了数十种不同类型的转换占位符,常见的如表 2.2 所示。

**表 2.2 常见的转换占位符**

| 转换类型 | 含义 |
|---|---|
| d | 带符号的十进制整数 |
| i | 带符号的十进制整数 |
| o | 不带符号的八进制 |
| u | 不带符号的十进制 |
| x | 不带符号的十六进制(小写) |
| X | 不带符号的十六进制(大写) |
| f、F | 十进制浮点数 |

续表

| 转换类型 | 含　义 |
| --- | --- |
| e | 用科学记数法表示的浮点数(小写) |
| E | 用科学记数法表示的浮点数(大写) |
| r | 字符串(使用 repr 转换任意 Python 对象) |
| s | 字符串(使用 str 转换任意 Python 对象) |

另一种格式化方式是使用 format()函数,这让输出格式更为清晰:

```
>>> str = "Hello world"
>>> length = len(str)
>>> print("字符串{0}的长度是:{1}".format(str, length))
字符串 Hello world 的长度是:11
```

在第 3 行代码中,{0}表示第 1 个占位符,{1}表示第 2 个占位符,以此类推;format()函数的第 1 个参数用于填充{0},第 2 个参数用于填充{1},以此类推。在使用 format()时并不区分填充占位符的变量的实际类型,这使得格式化工作更为轻松。format()的另一个好处是可以使用数组下标,例如:

```
>>> data = ["Xiaoming", 20]
>>> print("{0[0]} is {0[1]} years old.".format(data))
Xiaoming is 20 years old.
```

在以上代码中,{0[0]}表示使用 data 的第 1 个值,{0[1]}表示使用 data 的第 2 个值,因此分别把 data 的"Xiaoming"和 20 这两个值填充到前、后两个占位符,得到第 3 行所示的最终结果。

**2. title()函数**

例如:

```
>>> name = "lisa"
>>> print("I like " + name.title())
I like Lisa
```

在这个示例中,小写的字符串"lisa"存储到了变量 name 中。在 print()语句中,title()函数出现在这个变量的后面。方法函数 title()是 Python 可对数据执行的操作。在 name.title()中,name 后面的句点"."让 Python 对变量 name 执行 title()方

法指定的操作。每个函数后面都跟着一对括号,这是因为函数通常需要额外的信息来完成其工作。这种信息是在括号内提供的。title()函数不需要额外的信息,因此它后面的括号是空的。

title()以首字母大写的方式显示每个单词,即将每个单词的首字母都改为大写。这很有用,因为用户经常需要将名字视为信息。例如,用户可能希望程序将值 lisa、Lisa 和 LISA 视为同一个名字,并将它们都显示为 Lisa。

另外还有几个很有用的大小写处理方法。例如要将字符串改为全部大写或全部小写,可以像下面这样做:

```
>>> name = "lisa"
>>> print(name.upper())
LISA

>>> name = "LiSA"
>>> print(name.lower())
lisa
```

在存储数据时,lower()方法很有用。在很多时候无法依靠用户来提供正确的大小写,因此需要将字符串先转换为小写,再存储它们。这样以后需要显示这些信息时,再将其转换为最合适的大小写方式。

**3. rstrip()函数**

在程序中,额外的空白可能会令人迷惑。对程序员来说,“python”和“ python ”看起来几乎没什么两样,但对程序来说,它们却是两个不同的字符串。Python 能够发现“python”中额外的空白,并认为它是有意义的——除非告诉它不是这样的。

空白很重要,因为用户经常需要比较两个字符串是否相同。一个有代表性的示例是,在用户登录网站时检查其用户名。在一些简单得多的情形下,额外的空格也可能令人迷惑,所幸在 Python 中删除用户输入的数据中的多余空白很容易。

Python 能够找出字符串开头和末尾多余的空白,要确保字符串末尾没有空白,可以使用 rstrip()函数。

```
>>> language = " python "
>>> language
' python '
```

```
>>> language.rstrip()
' python'
>>> language
' python '
```

存储在变量 language 中的字符串末尾包含多余的空白(见第 1 行)。用户在终端会话中向 Python 询问这个变量的值时可以看到末尾的空格(见第 3 行)。对变量 language 调用 rstrip()方法后,这个多余的空格被删除了(见第 4 行)。然而,这种删除只是暂时的,接下来再次询问 language 的值时,用户会发现这个字符串与输入时一样,依然包含多余的空白(见第 7 行)。

如果要永久删除这个字符串中的空白,必须将删除操作的结果存回到变量中:

```
>>> language = ' python '
>>> language = language.rstrip()
>>> language
' python'
```

为了删除这个字符串中的空白,用户需要将其末尾的空白去除,再将结果存回到原来的变量中(见第 2 行)。在编程中经常需要修改变量的值,再将新值存回到原来的变量中,这就是变量的值可能随程序的运行或用户输入的数据而发生变化的原因。

用户还可以去除字符串开头的空白,或同时去除字符串两端的空白。为此,可以分别使用 lstrip()和 strip()方法:

```
>>> language = ' python '
>>> language.rstrip()
' python'
>>> language.lstrip()
'python '
>>> language.strip()
'python'
```

在这个示例中,首先创建了一个开头和末尾都有空白的字符串,接下来分别删除末尾(见第 2 行)、开头(第 4 行)和两端(第 6 行)的空格。尝试使用这些函数有助于用户熟悉字符串操作。在实际程序中,这些函数经常用于在存储用户输入前对其进行清理。

## 本章小结

在本章中大家学习了在 Python 中如何创建合法的变量名、给变量赋值，以及各种数据类型和 Python 中常用的函数。通过本章的学习，读者还理解了分支结构控制语句的基本用法，if 语句允许在脚本中使用一个或多个条件来检查数据，可以添加 else 语句，在条件失败时以提供另一条逻辑路径；可以通过在 if 语句中使用一个或多个 elif 来扩展，将 elif 语句串联到一起来不断地比较额外的值。另外，本章还介绍了如何创建 for 循环和 while 循环，并讲解了嵌套循环。

## 习题

请完成下面的练习，在做每个练习时都要编写一个独立的程序，在保存每个程序时使用符合标准 Python 约定的文件名，即使用小写字母和下画线，例如 simple_message. py 和 simple_messages. py。

（1）简单消息：将一条消息存储到变量中，再将其打印出来。

（2）多条简单消息：将一条消息存储到变量中，将其打印出来；再将变量的值修改为一条新消息，并将其打印出来。

# 第 3 章



# 数据结构与函数设计

**本章学习目标：**

- 熟练掌握序列的基本概念
- 熟练掌握列表、元组、字典、字符串的概念和各种用法
- 熟练掌握各种序列类型之间的转化
- 了解集合的基本概念和用法
- 熟练掌握自定义函数的设计和使用
- 深入了解各类参数以及传递过程

本章主要介绍两方面内容，一是常用的数据结构，二是函数设计。在数据结构方面，先介绍序列的基本概念，然后介绍各种序列类型，包括列表、元组、字符串和字典，最后讲解集合的概念和用法；在函数设计方面，先介绍函数的定义，接着对函数的返回值和形参、实参、默认参数、关键参数、可变长度参数、序列参数等各类参数进行介绍，由此完成对函数的比较细致、全面的讲解。

## 3.1 序列

在 Python 中，最基本的数据结构是序列(sequence)。序列中的每个元素被分配一个序号，即元素的位置，也称为索引。第 1 个索引是 0，第 2 个是 1，以此类推。序列

中的最后一个元素标记为－1，倒数第2个元素为－2，以此类推。

Python中包含6种内建的序列，即列表、元组、字符串、Unicode字符串、buffer对象和xrange对象。本章重点讨论列表和元组，列表和元组的主要区别在于列表可以修改，而元组不能。

所有序列类型都可以进行某些特定的操作，这些操作包括索引（indexing）、分片（sliceing）、加（adding）、乘（multiplying）以及检查某个元素是否属于序列的成员（成员资格）。除此之外，Python还有计算序列长度、找出最大元素和最小元素的内建函数。

### 3.1.1 列表


视频讲解

列表由一系列按特定顺序排列的元素组成。用户可以创建包含字母表中所有字母、数字0～9或所有家庭成员姓名的列表；也可以将任何内容加入到列表中，其中的元素之间可以没有任何关系。鉴于列表通常包含多个元素，给列表指定一个表示复数的名称（例如letters、digits或names）是个不错的主意。在Python中，用方括号（[]）来表示列表，并用逗号来分隔其中的元素。下面是一个简单的列表示例，这个列表包含几种自行车。

**【例3-1】** 列表实例。

```
bicycles.py
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles)
```

如果让Python将列表打印出来，Python将打印列表的内部表示，包括方括号，即['trek', 'cannondale', 'redline', 'specialized']，但这不是要让用户看到的输出。下面来学习如何访问列表元素。

列表是有序集合，因此要访问列表中的任何元素，只需将该元素的位置或索引告诉Python即可。如果要访问列表元素，可以先指出列表的名称，再指出元素的索引，并将其放在方括号内。例如，代码“bicycles = [ 'trek ', 'cannondale ', 'redline ', 'specialized']print(bicycles[0])”从列表bicycles中提取第一款自行车。当用户请求获取列表元素时，Python只返回该元素（即trek），而不包括方括号和引号，这正是想要看到的整洁、干净的输出。用户还可以对任何列表元素调用第2章介绍的字符串

方法，例如可以使用 title()方法让元素'trek'的格式更整洁，即“ bicycles=['trek', 'cannondale', 'redline', 'specialized'] print(bicycles[0]. title())”，这个示例的输出与前一个示例相同，只是首字母 T 是大写的。

例 3-2 为 Python 的列表操作。

**【例 3-2】** 列表操作实例。

```
sample_list = [ 'a', 'b',0,1,3]
```

得到列表中的某一个值：

```
value_start = sample_list[0]
end_value = sample_list[-1]
```

删除列表的第 1 个值：

```
del sample_list[0]
```

在列表中插入一个值：

```
sample_list.insert(0,'sample value')
```

得到列表的长度：

```
list_length = len(sample_list)
```

列表遍历：

```
for element in sample_list:print(element)
```

下面是 Python 列表的高级操作和技巧。

产生一个数值递增列表：

```
num_inc_list = range(30)            #返回列表[0,1,2,...,29]
```

用某个固定值初始化列表：

```
initial_value = 0
list_length = 5
```

```
sample_list = [initial_value for i in range(10)]
sample_list = [initial_value] * list_length
# sample_list == [0,0,0,0,0]
```

list 的方法：

```
L.append(var)          #追加元素
L.insert(index,var)    #在指定位置插入元素
L.pop(var)             #返回最后一个元素,并从 list 中将其删除
L.remove(var)          #删除第一次出现的该元素
L.count(var)           #该元素在列表中出现的个数
L.index(var)           #该元素的位置,无则抛出异常
L.extend(list)         #追加 list,即合并 list 到 L 上
L.sort()               #排序
L.reverse()            #倒序
```

list 操作符：、+、*，关键字 del：

```
a[1:]                  #片段操作符,用于子 list 的提取
[1,2] + [3,4]          #为[1,2,3,4],同 extend()
[2] * 4                #为[2,2,2,2]
del L[1]               #删除指定下标的元素
del L[1:3]             #删除指定下标范围的元素
```

list 的复制：

```
L1 = L                 #L1 为 L 的别名,用 C 语言来说就是指针地址相同,对 L1 操作即对 L 操作
                       #函数参数就是这样传递的
L1 = L[:]              #L1 为 L 的克隆,即另一个副本
```

### 3.1.2 元组

创建元组（即常量数组）：

```
tuple = ('a', 'b', 'c', 'd', 'e')
```

元组可以用 list 的[]操作符提取元素，但不能直接修改元素。

元组的操作：索引、切片、连接、重复。

**【例 3-3】** 元组操作实例。

```
t = ("fentiao", 5, "male")
#正向索引
print(t[0])
#反向索引
print(t[-1])
#元组嵌套时元素的访问
t1 = ("fentiao", 5, "male", ("play1", "play2", "play3") )
print(t1[3][1])
#切片
print(t[:2])
#逆转元组元素
print(t[::-1])
#连接
print(t + t1)
#重复
t * 3
```

### 3.1.3　字符串

字符串是 Python 语言中的一种数据类型。字符串由任意字符构成，一个字符可能是一个字母、数值、符号或者标点符号。字符串是用来记录文本信息的，它们在 Python 中作为序列，也就是说一个包含其他对象的有序集合。序列中的元素包含了一个从左到右的顺序，序列中的元素根据它们的相对位置进行存储和读取。从严格意义上说，字符串是单个字符的字符串的序列。

例如：

```
str = "Hello My friend"
```

字符串是一个整体，如果用户想直接修改字符串的某一部分，则是不可能的，但能够读出字符串的某一部分。

子字符串的提取：

```
str[:6]
```

字符串包含判断操作符：in、not in

```
"He" in str
"she" not in str
```

string 模块提供的方法：

```
S.find(substring, [start [,end]])      #可指定范围查找子串,返回索引值,否则返回-1
S.rfind(substring,[start [,end]])      #反向查找
S.index(substring,[start [,end]])      #同 find(),只是找不到产生 ValueError 异常
S.rindex(substring,[start [,end]])     #同上,反向查找
S.count(substring,[start [,end]])      #返回找到子串的个数
S.lowercase()                          #首字母小写
S.capitalize()                         #首字母大写
S.lower()                              #转小写
S.upper()                              #转大写
S.swapcase()                           #大小写互换
S.split(str, ' ')                      #将 string 转 list,以空格切分
S.join(list, ' ')                      #将 list 转 string,以空格连接
```

处理字符串的内置函数：

```
len(str)                               #串长度
cmp("my friend", str)                  #字符串比较,第一个大,返回 1
max('abcxyz')                          #寻找字符串中最大的字符
min('abcxyz')                          #寻找字符串中最小的字符
```

string 的转换：

```
float(str)                             #变成浮点数,float("1e-1")的结果为 0.1
int(str)                               #变成整型,int("12") 的结果为 12
int(str,base)                          #变成 base 进制整型数,int("11",2)的结果为 2
long(str)                              #变成长整型
long(str,base)                         #变成 base 进制长整型
```

字符串的格式化(注意其转义字符,大多如 C 语言)：

```
str_format % (参数列表)      #参数列表是以 tuple 的形式定义的,即不可以在运行中改变
```

**【例 3-4】** 字符串操作实例。

```
>>> print("%s's height is %dcm" % ("My brother", 180))
My brother's height is 180cm
```

### 3.1.4　列表与元组之间的转换

使用 list()和 tuple()函数进行元组和列表的相互转换。

```
tuple(ls)
list(ls)
```

## 3.2　字典



视频讲解

字典是一种通过名字或者关键字引用的数据结构,其键可以是数字、字符串、元组,这种结构类型也称为映射。字典类型是 Python 中唯一内建的映射类型。

### 3.2.1　创建字典

(1) 直接创建字典:

```
d = {'one':1,'two':2,'three':3}
```

(2) 通过 dict()创建字典:

```
#_*_ coding:utf-8 _*_
items = [('one',1),('two',2),('three',3),('four',4)]
d = dict(items)
print(d)
```

(3) 通过关键字创建字典:

```
#_*_ coding:utf-8 _*_
d = dict(one = 1,two = 2,three = 3)
print(d)
print(d['one'])
print(d['three'])
```

(4) 字典的格式化字符串:

```
#_*_ coding:utf-8 _*_
d = {'one':1,'two':2,'three':3,'four':4}
```

```
print(d)
print("three is %(three)s." % d)
```

### 3.2.2 字典的方法

每一个元素是 pair，包含 key、value 两部分。key 是 integer 或 string 类型，value 是任意类型。键是唯一的，字典只认最后一个赋的键值。

dictionary 的方法：

```
D.get(key, 0)           #同 dict[key],如果指定键的值不存在,返回默认值
D.has_key(key)          #有该键返回 True,否则返回 False
D.keys()                #返回字典键的列表
D.values()
D.items()
D.update(dict2)         #增加合并字典
D.popitem()             #得到一个 pair,并从字典中删除它,若已空则抛出异常
D.clear()               #清空字典,同 del dict
D.copy()                #复制字典
D.cmp(dict1,dict2)      #比较字典(优先级为元素个数、键值大小)
                        #第一个大返回 1,小返回-1,一样返回 0
```

dictionary 的复制：

```
dict1 = dict            #别名
dict2 = dict.copy()     #克隆,即另一个副本
```

### 3.2.3 列表、元组与字典之间的转换

这三者之间的转换并不复杂，但字典的转换由于有 key 的关系，所以其他二者不能转换为字典。

(1) 对元组进行转换：

```
>>> fruits = ('apple', 'banana', 'orange')          #元组转换为列表
>>> list(fruit)
```

(2) 对列表的转换：

```
>>> fruit_list = ['apple', 'banana', 'orange']      #列表转换为元组
>>> tuple(fruit_list)
```

(3) 对字典的转换：用户可以使用 tuple()和 list()函数将字典转换为元组和列表,但要注意这里转换后和转换前的元素顺序是不同的,因为字典类似于散列,列表类似于链表,元组类似于列表,只是元素无法改变,所以要把散列转换为链表而顺序不变是不可行的。此时可以借助于有序字典(OrderedDict),有序字典是字典的子类,它可以记住元素添加的顺序,从而得到有序的字典。对于有序字典这里不深入探讨,只给出普通字典的例子做参考,代码如下：

**【例 3-5】** 字典转换例子。

```
>>> fruit_dict = {'apple':1, 'banana':2, 'orange':3}
>>> tuple(fruit_dict)               #将字典的 key 转换为元组
>>> tuple(fruit_dict.value())       #将字典的 value 转换为元组
>>> list(fruit_dict)                #将字典的 key 转换为列表
>>> list(fruit_dict.value())        #将字典的 value 转换为列表
```

## 3.3 集合

### 3.3.1 集合的创建

在 Python 中集合由内置的 set 类型定义,如果要创建集合,需要将所有项(元素)放在花括号({})内,以逗号(,)分隔。

**【例 3-6】** 集合实例。

```
>>> s = {'P', 'y', 't', 'h', 'o', 'n'}
>>> type(s)
<class 'set'>
```

集合可以有任意数量的元素,它们可以是不同的类型(例如数字、元组、字符串等),但是集合不能有可变元素(例如列表、集合或字典)：

```
>>> s = {1, 2, 3}                              #整型的集合
>>> s = {1.0, 'Python', (1, 2, 3)}             #混合类型的集合
>>> s = set(['P', 'y'])                        #从列表创建
>>> s = {1, 2, [3, 4]}                         #不能有可变元素
TypeError: unhashable type: 'list'
```

创建空集合比较特殊,在 Python 中空花括号({})用于创建空字典。如果要创建

一个没有任何元素的集合，使用 set()函数(不要包含任何参数)：

```
>>> d = {}                          #空字典
>>> type(d)
<class 'dict'>
>>> s = set()                       #空集合
>>> type(s)
<class 'set'>
```

回顾数学的相关知识，发现集合具有以下特性。

(1) 无序性：在一个集合中，每个元素的地位都是相同的，元素之间是无序的。在集合上可以定义序关系，在定义了序关系之后元素之间就可以按照序关系排序，但就集合本身的特性而言，元素之间没有必然的序。

(2) 互异性：在一个集合中，任何两个元素都认为是不相同的，即每个元素只能出现一次。有时需要对同一元素出现多次的情形进行刻画，此时可以使用多重集，其中的元素允许出现多次。

(3) 确定性：给定一个集合，任意一个元素，该元素或者属于或者不属于该集合，二者必是其一，不允许有模棱两可的情况出现。

当然，Python 中的集合也具有这些特性。例如：

```
#无序性
>>> s = set('Python')
>>> s
{'y', 'n', 'h', 'o', 'P', 't'}
>>> s[0]                            #不支持索引
TypeError: 'set' object does not support indexing
#互异性
>>> s = set('Hello')
>>> s
{'e', 'H', 'l', 'o'}
#确定性
>>> 'l' in s
True
>>> 'P' not in s
True
```

**注意**：由于集合是无序的，所以索引没有任何意义。也就是说，无法使用索引或切片访问或更改集合元素。

### 3.3.2 集合的运算

集合之间也可以进行数学集合运算(例如并集、交集等),可用相应的操作符或方法来实现。考虑 A、B 两个集合,进行以下操作。

**【例 3-7】** 集合运算实例。

```
>>> A = set('abcd')
>>> B = set('cdef')
```

#### 1. 子集

子集为某个集合中一部分的集合,故也称部分集合。

使用操作符“<”执行子集操作,同样,也可以使用 issubset()方法完成。例如:

```
>>> C = set('ab')
>>> C < A
True
>>> C < B
False
>>> C.issubset(A)
True
```

#### 2. 并集

一组集合的并集是这些集合的所有元素构成的集合,而不包含其他元素。

使用操作符“|”执行并集操作,同样,也可以使用 union()方法完成。例如:

```
>>> A | B
{'e', 'f', 'd', 'c', 'b', 'a'}
>>> A.union(B)
{'e', 'f', 'd', 'c', 'b', 'a'}
```

#### 3. 交集

两个集合 A 和 B 的交集是含有所有既属于 A 又属于 B 的元素且没有其他元素的集合。

使用操作符“&”执行交集操作,同样,也可以使用 intersection()方法完成。例如:

```
>>> A & B
{'d', 'c'}
>>> A.intersection(B)
{'d', 'c'}
```

#### 4. 差集

A 与 B 的差集是所有属于 A 但不属于 B 的元素构成的集合。

使用操作符“-”执行差集操作，同样，也可以使用 difference()方法完成。例如：

```
>>> A - B
{'b', 'a'}
>>> A.difference(B)
{'b', 'a'}
```

#### 5. 对称差

两个集合的对称差是只属于其中一个集合，而不属于另一个集合的元素组成的集合。

使用操作符“^”执行对称差操作，同样，也可以使用 symmetric_difference()方法完成。例如：

```
>>> A ^ B
{'b', 'e', 'f', 'a'}
>>> A.symmetric_difference(B)
{'b', 'e', 'f', 'a'}
```

#### 6. 更改集合

虽然集合中不能有可变元素，但集合本身是可变的。也就是说，可以添加或删除其中的元素。

用户可以使用 add()方法添加单个元素，使用 update()方法添加多个元素，update()可以使用元组、列表、字符串或其他集合作为参数。例如：

```
>>> s = {'P', 'y'}
>>> s.add('t')                          #添加一个元素
>>> s
{'P', 'y', 't'}
>>> s.update(['h', 'o', 'n'])           #添加多个元素
>>> s
{'y', 'o', 'n', 't', 'P', 'h'}
```

```
>>> s.update(['H', 'e'], {'l', 'l', 'o'})          #添加列表和集合
>>> s
{'H', 'y', 'e', 'o', 'n', 't', 'l', 'P', 'h'}
```

在所有情况下元素都不会重复。

**7. 从集合中删除元素**

用户可以使用 discard()和 remove()方法删除集合中特定的元素。

两者之间唯一的区别在于如果集合中不存在指定的元素，使用 discard()结果保持不变，但在这种情况下 remove()会引发 KeyError。例如：

```
>>> s = {'P', 'y', 't', 'h', 'o', 'n'}
>>> s.discard('t')                          #去掉一个存在的元素
>>> s
{'y', 'o', 'n', 'P', 'h'}
>>> s.remove('h')                           #删除一个存在的元素
>>> s
{'y', 'o', 'n', 'P'}
>>> s.discard('w')                          #去掉一个不存在的元素(正常)
>>> s
{'y', 'o', 'n', 'P'}
>>> s.remove('w')                           #删除一个不存在的元素(引发错误)
KeyError: 'w'
```

类似地，用户可以使用 pop()方法删除和返回一个项目，还可以使用 clear()方法删除集合中的所有元素。例如：

```
>>> s = set('Python')
>>>
>>> s.pop()                                  #随机返回一个元素
'y'
>>>
>>> s.clear()                                #清空集合
```

**注意**：集合是无序的，所以无法确定哪个元素将被 pop，完全随机。

### 3.3.3　集合的方法

使用 dir()来查看方法列表：

```
>>> dir(set)
['__and__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__iand__', '__init__', '__ior__', '__isub__', '__iter__', '__ixor__', '__le__', '__len__', '__lt__', '__ne__','__new__', '__or__', '__rand__', '__reduce__', '__reduce_ex__', '__repr__', '__ror__','__rsub__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__xor__', 'add', 'clear', 'copy', 'difference', 'difference_update', 'discard', 'intersection', 'intersection_update', 'isdisjoint', 'issubset', 'issuperset', 'pop', 'remove', 'symmetric_difference', 'symmetric_difference_update', 'union', 'update']
```

可以看到有表 3.1 所示的方法可用。

**表 3.1　集合的方法**

| 方　法 | 描　述 |
|---|---|
| add() | 将元素添加到集合中 |
| clear() | 删除集合中的所有元素 |
| copy() | 返回集合的浅复制 |
| difference() | 将两个或多个集合的差集作为一个新集合返回 |
| difference_update() | 从一个集合中删除另一个集合的所有元素 |
| discard() | 删除集合中的一个元素(如果元素不存在,则不执行任何操作) |
| intersection() | 将两个集合的交集作为一个新集合返回 |
| intersection_update() | 用自己和另一个的交集来更新这个集合 |
| isdisjoint() | 如果两个集合有一个空交集,返回 True |
| issubset() | 如果另一个集合包含这个集合,返回 True |
| issuperset() | 如果这个集合包含另一个集合,返回 True |
| pop() | 删除并返回任意的集合元素(如果集合为空,会引发 KeyError) |
| remove() | 删除集合中的一个元素(如果元素不存在,会引发 KeyError) |
| symmetric_difference() | 将两个集合的对称差作为一个新集合返回 |
| symmetric_difference_update() | 用自己和另一个的对称差来更新这个集合 |
| union() | 将集合的并集作为一个新集合返回 |
| update() | 用自己和另一个的并集来更新这个集合 |

其中一些方法在上述示例中已经用过,如果有些方法用户不会用,可使用 help()函数查看其用途及详细说明。

### 1. 集合与内置函数

表 3.2 中的内置函数通常作用于集合执行不同的任务。

表 3.2　集合的内置函数

| 函　数 | 描　述 |
|---|---|
| all() | 如果集合中的所有元素都是 True(或者集合为空),则返回 True |
| any() | 如果集合中的所有元素都是 True,则返回 True；如果集合为空,则返回 False |
| enumerate() | 返回一个枚举对象,其中包含了集合中所有元素的索引和值(配对) |
| len() | 返回集合的长度(元素个数) |
| max() | 返回集合中的最大项 |
| min() | 返回集合中的最小项 |
| sorted() | 从集合中的元素返回新的排序列表(不排序集合本身) |
| sum() | 返回集合的所有元素之和 |

### 2. 不可变集合

frozenset 是一个具有集合特征的新类,但是一旦分配,它里面的元素就不能更改。这一点和元组非常类似：元组是不可变的列表,frozenset 是不可变的集合。集合是 unhashable 的,因此不能用作字典的 key；而 frozenset 是 hashable 的,可以用作字典的 key。用户可以使用 frozenset()函数创建 frozenset,例如：

```
>>> s = frozenset('Python')
>>> type(s)
<class 'frozenset'>
```

frozenset 也提供了一些方法,和 set 中的类似,同样使用 dir()查看：

```
>>> dir(frozenset)
['__and__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '
__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__
', '__ne__', '__new__', '__or__', '__rand__', '__reduce__', '__reduce_ex__','__repr__', '__ror__', '__rsub
__', '__rxor__', '__setattr__', '__sizeof__','__str__', '__sub__', '__subclasshook__', '__xor__', 'copy',
'difference', 'intersection', 'isdisjoint', 'issubset', 'issuperset', 'symmetric_difference', 'union']
```

由于 frozenset 是不可变的,所以没有添加或删除元素的方法。

## 3.4　函数的定义

函数是一段按逻辑组织好的、可重复使用来实现单一或者相关联功能的代码,使用函数能有效地提高应用的模块性和代码的重复利用率。在 Python 中已提供了许多内建函数,例如 print()。用户也可以自己创建函数来满足特定的需求,这种函数

被称为用户自定义函数。本节的主要内容就是用户自定义函数的设计和实现。

在 Python 中用户可以根据自己的需求自定义函数,但是在定义函数的过程中必须遵循以下几条简单的规则:

(1) 函数代码块以 def 关键字开头,后接函数标识符名称和圆括号"()"。

(2) 任何传入参数和自变量必须放在圆括号之内,圆括号之间用于定义参数。

(3) 函数的第 1 行语句可以选择性地使用文档字符串(用于存放函数说明)。

(4) 函数内容以冒号起始,并且缩进。

(5) "return [表达式]"结束函数,选择性地返回一个值给调用方,不带表达式的 return 相当于返回 None。

下面是定义函数的一般语法:

```
>>> def functionname(parameters):
        """函数_文档字符串"""
            function_suite
            return [expression]
```

在默认情况下,参数值和参数名称都是按函数声明中参数定义时的顺序匹配起来的。下面是自定义函数的一个简单实例,用来打印简单的问候语。

**【例 3-8】** 自定义函数实例。

```
>>> def greeting():
 """显示简单的问候语"""
        print("Hello!Python world")
>>> greeting()
Hello!Python world
```

这个实例演示了最简单的自定义函数结构。第 1 行代码使用关键字 def 来告诉 Python 接下来要定义一个函数。这是函数定义向 Python 指出了函数名,还可能在括号内指出函数为完成其任务需要哪些参数。在这里函数名为 greeting(),它不需要任何参数就能完成其工作,因此括号内是空的(即便如此,括号也必不可少)。最后定义以冒号结尾。

紧跟在"def greeting():"后面的所有缩进行构成了函数体。引号内的文本被称为注释,描述了该函数是用来做什么的。文档字符串用三引号括起,Python 使用它们来生成有关程序中函数的文档。

代码行 print("Hello! Python world")是函数体内唯一的一行代码,greeting()只做一项工作,即打印“Hello! Python world”。

如果要使用这个函数,可调用它,函数调用让 Python 执行函数的代码。如果要调用函数,可依次指定函数名以及用括号括起来的必要的参数,如第 4 行代码所示。由于这个函数不需要任何参数,因此在调用它时只需要输入 greeting()即可,和预期的一样,它打印“Hello! Python world”。

### 3.4.1　函数的调用



视频讲解

定义一个函数只是给了函数一个名称,以及指定了函数中应该包含哪些参数和代码块结构。当一个函数的基本结构完成以后,就可以在另一个函数里调用执行,当然也可以直接从 Python 提示符执行。下面通过实例来介绍如何调用自定义函数。

**【例 3-9】**　函数调用实例 1。

```
#定义函数
>>> def printme( strings ):
 """打印任何传入的字符串"""
     print(strings)
     return

#调用函数
>>> printme("我要调用用户自定义函数!")
>>> printme("再次调用同一函数")
```

以上实例的输出结果:

```
我要调用用户自定义函数!
再次调用同一函数
```

### 3.4.2　形参与实参

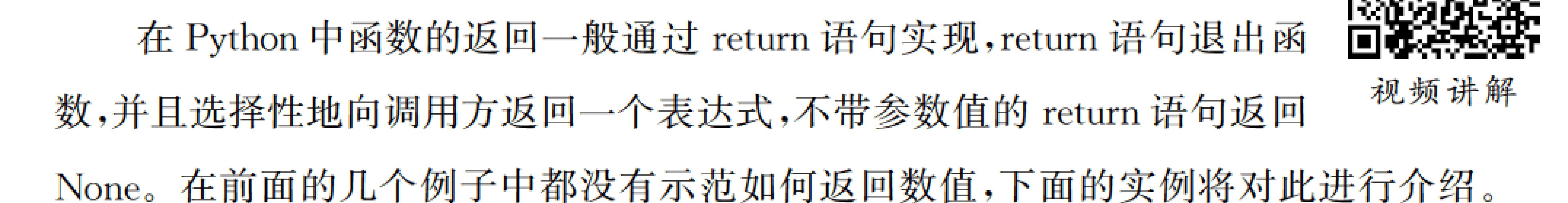
在前面定义 greeting()函数时并没有指定参数,现在假设给函数指定参数 username,调用这个函数并提供这种信息(人名),它将打印相应的问候语。

这样,在 greeting()函数的定义中变量 username 就是一个形参(函数完成其工作所需的一项信息)。在函数的调用代码 greeting('Tom')中,值'Tom'是一个实参。实

参是调用函数时传递给函数的信息。在调用函数时将要让函数使用的信息放在括号内。在 greeting('Tom')中将实参'Tom'传递给了 greeting()函数,这个值被存储在形参 username 中。

### 3.4.3 函数的返回

视频讲解



在 Python 中函数的返回一般通过 return 语句实现,return 语句退出函数,并且选择性地向调用方返回一个表达式,不带参数值的 return 语句返回 None。在前面的几个例子中都没有示范如何返回数值,下面的实例将对此进行介绍。

**【例 3-10】** 函数调用实例 2。

```
#可写函数说明
>>> def sum(arg1, arg2):
    #返回两个参数的和
      total = arg1 + arg2
      print("函数内 : ", total)
      return total

#调用 sum()函数
>>> total = sum(10, 20)
```

以上实例的输出结果:

```
函数内 : 30
```

### 3.4.4 位置参数

视频讲解

在调用函数时,Python 必须将函数调用中的每个实参都关联到函数定义中的每个形参。因此,最简单的关联方式是基于实参的顺序,这种关联方式被称为位置参数。简单地说,就是在给函数传参数时按照顺序依次传值。下面通过实例进行介绍。

**【例 3-11】** 位置参数实例。

```
>>> def power(m, n):
        result = 1
        while n > 0:
            n = n-1
            result = result * m
        return result
```

调用函数并输出结果：

```
>>> print(power(4,3))
64
```

在 power(m,n)函数中有两个参数，即 m 和 n，这两个参数都是位置参数，在调用的时候传入的两个值按照顺序依次赋给 m 和 n。

### 3.4.5　默认参数与关键字参数



视频讲解

所谓默认参数，就是在写函数的时候直接给参数传默认的值，在调用的时候默认参数已经有值，这样就不用再传值了，最大的好处就是降低调用函数的难度。

修改例 3-11，见例 3-12。

**【例 3-12】** 默认参数实例。

```
>>> def power(m, n = 3):
        result = 1
            while n > 0:
            n = n-1
            result = result * m
        return result
```

调用函数并输出结果：

```
>>> print(power(4))
64
```

在修改后的函数中，对第 2 个形参 n 设置了一个默认值 3，在之后的函数调用中不提供该函数的实参，但函数仍旧能正常运行，因为在这个函数中 n 已经是一个默认参数。在设置默认参数时需要注意两点：一是必选参数在前，默认参数在后，否则 Python 解释器会报错；二是默认参数一定要指向不变对象。

关键字参数和函数调用关系紧密，在函数调用时可以通过使用关键字参数来确定传入的参数值。其最显著的特征就是使用关键字参数将允许函数调用时参数的顺序与声明时不一致，因为 Python 解释器能够用参数名匹配参数值。

下面通过实例来介绍关键字参数。

**【例 3-13】** 关键字参数实例 1。

```
#可写函数说明
>>> def printme(strings):
      """打印任何传入的字符串"""
          print(strings)
          return

#调用 printme()函数
>>> printme(strings = "My string")
```

以上实例的输出结果：

```
My string
```

下例能将关键字参数的顺序不重要展示得更清楚。

**【例 3-14】** 关键字参数实例 2。

```
#可写函数说明
>>>def printinfo(name, age):
     """打印任何传入的字符串"""
      print("Name: ", name)
      print("Age ", age)
      return

#调用 printinfo()函数
>>> printinfo( age = 40, name = "James")
```

以上实例的输出结果：

```
Name: James
Age 40
```

## 3.4.6 可变长度参数

在 Python 函数中还可以定义可变长度参数。顾名思义，可变长度参数就是所传入参数的个数是可变的，可以是一个、两个到任意个，也可以是 0 个。

和前面 3 种参数不同，可变长度参数在声明时不会全部命名。其基本语法如下：

```
>>> def functionname([formal_args,] * var_args_tuple):
    """函数_文档字符串"""
        function_suite
        return [expression]
```

注意，加了星号（*）的变量名会存放所有未命名的变量参数。

可变长度参数的实例如下：

**【例 3-15】** 可变长度参数实例。

```
#可写函数说明
>>> def printinfo(arg1, *vartuple):
    """打印任何传入的参数"""
        print("输出：")
        print(arg1)
        for var in vartuple:
            print(var)
            return

#调用 printinfo()函数
>>> printinfo(15)
>>> printinfo(75, 65, 55)
```

以上实例的输出结果：

```
15
75
65
55
```

## 本章小结

本章的主要内容分为两个部分。在第一部分介绍了 Python 中常见的数据结构，包括序列（例如列表、元组）、映射（例如字典）和集合等，主要包括序列的基本概念，序列的各种方法，字符串的两种重要使用方式（字符串格式化与字符串方法），利用字典格式化字符串以及字典的用法，集合的创建、运算和方法。

在第二部分介绍了自定义函数的基本语法、几种形式的函数参数、返回值。自定义函数由 def 开头，接下来确定函数名、形参的类型数量，还要加冒号，最后缩进写入函数体。形参是可选的，可有可无；同样，函数可以有返回值，也可以没有返回值，其主要通过 return 语句返回，也就是通过 return 语句将程序的控制权返回给函数的调用者。Python 的函数具有非常灵活的参数形态，既可以实现简单的调用，又可以传入非常复杂的参数。函数参数可以作为位置参数或者关键字参数进行传递，并且可

以使用默认参数传递，以及使用可变长度参数进行传递。

## 习题

1. 掌握并比较不同数据类型的异同。

2. 利用循环创建一个包含 100 个偶数的列表，并且计算该列表的和与平均值。请分别使用 while 和 for 循环实现。

3. 编写一个函数实现根据本金、年利率、存款年限计算得到本金和利息的功能，调用这个函数计算 10 000 元本金在银行以 5.5%的年利率存 5 年后获得的本金和利息。在该题中按复利计算利息。

4. 编写一个函数实现判断一个输入的数字是否为奇数的功能。

5. 编写一个名为 MakeTshirt()的函数，它接受一个尺码以及要印到 T 恤上的字样。这个函数应打印一个句子，概要地说明 T 恤的尺码和字样。请分别使用位置参数和关键字参数调用这个函数来制作一件 T 恤。

# 第 4 章



# 类与对象

**本章学习目标：**

- 深刻理解 Python 中类、对象的概念，掌握它们的构造和使用
- 熟练掌握 Python 面向对象的构造函数和析构函数，以及运算符的重载
- 理解 Python 类的继承和组合
- 熟练掌握 Python 异常处理机制和内置异常类
- 熟练掌握 Python 自定义异常的方法

本章首先从 Python 类和对象的定义开始讲解，详细介绍类的属性、方法，以及构造函数和析构函数，然后介绍面向对象的方法以及类的两种重用技术——继承和组合，最后向读者介绍异常的基本概念和处理机制，同时介绍自定义异常的方法、With 语句和断言。

## 4.1 面向对象

面向对象编程的英文全称为 Object Oriented Programming，简称 OOP，它是一种程序设计思想。OOP 把对象作为程序的基本单元，一个对象包含了数据和操作数据

的函数。

面向对象的程序设计把计算机程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。

例如，程序中包含一个 customer 对象和一个 account 对象，而 customer 对象可能会向 account 对象发送一个消息，查询其银行账目。每个对象都包含数据以及操作这些数据的代码，即使它们不了解彼此的数据和代码的细节，对象之间依然可以相互作用，所要了解的只是对象能够接受的消息的类型，以及对象返回的响应的类型，虽然不同的人会以不同的方法实现它们。

### 4.1.1 面向对象编程


视频讲解

面向对象编程的优点是易维护、易复用、易扩展，由于面向对象有封装、继承、多态的特性，可以设计出低耦合的系统，使系统更加灵活、更加易于维护。

在 Python 中所有数据类型都可以视为对象，当然用户也可以自定义对象。自定义的对象数据类型就是面向对象中的类(Class)的概念。

如果采用面向对象的程序设计思想，大家首先思考的不是程序的执行流程，而是 Student 这种数据类型应该被视为一个类，这个类拥有 name 和 score 两个属性(Property)。如果要打印一个学生的成绩，首先必须创建出这个学生类对应的对象，然后给对象发一个 print_score 消息，让对象自己把自己的数据打印出来。具体如下。

**【例 4-1】** 面向对象示例。

```
class Student(object):
    def __init__(self, name, score):
        self.name = name
        self.score = score
    def print_score(self):
        print("%s: %s" % (self.name, self.score))
```

给对象发消息实际上就是调用对象对应的关联函数，一般称之为对象的方法(Method)。面向对象的程序写出来就像这样：

```
bart = Student("Bart Simpson", 59)
lisa = Student("Lisa Simpson", 87)
```

```
bart.print_score()
lisa.print_score()
```

面向对象类(Class)和实例(Instance)的设计思想是从自然界中来的。Class 是一种抽象概念,比如这里定义的 Student 是指学生这个概念,而实例(Instance)是一个个具体的 Student,比如 Bart Simpson 和 Lisa Simpson 是两个具体的 Student。

所以,面向对象的设计思想是抽象出 Class,根据 Class 创建 Instance。面向对象的抽象程度要比函数高,因为一个 Class 既包含数据又包含操作数据的方法。

### 4.1.2　类的抽象与封装


视频讲解

对象包含数据以及操作这些数据的代码,一个对象包含的所有数据和代码可以通过类构成一个用户定义的数据类型。事实上,对象就是类类型(class type)的变量,一旦定义了一个类,就可以创建这个类的多个对象,每个对象与一组数据相关,而这组数据的类型在类中定义。因此,一个类就是具有相同类型的对象的抽象,例如杧果、苹果和橘子都是 fruit 类的对象。类是用户定义的数据类型,但是在一个程序设计语言中,它和内建的数据类型行为相同。比如创建一个类对象的语法和创建一个整数对象的语法一模一样。

把数据和函数装在一个单独的单元(称为类)的行为称为封装。数据封装是类最典型的特点。数据不能被外界访问,只能被封装在同一个类中的函数访问。这些函数提供了对象数据和程序之间的接口。避免数据被程序直接访问的概念被称为"数据隐藏"。

抽象指仅表现核心的特性而不描述背景细节的行为。类使用了抽象的概念,并且被定义为一系列抽象的属性,例如尺寸、重量和价格,以及操作这些属性的函数。类封装了将要被创建的对象的所有核心属性。因为类使用了数据抽象的概念,所以它们被称为抽象数据类型(ADT)。

封装机制将数据和代码捆绑到一起,避免了外界的干扰和不确定性。它同样允许创建对象。简单地说,一个对象就是一个封装了数据和操作这些数据的代码的逻辑实体。

在一个对象内部,某些代码和(或)某些数据可以是私有的,不能被外界访问。通过这种方式,对象对内部数据提供了不同级别的保护,以防止程序中无关的部分意外

地改变或错误地使用了对象的私有部分。

简而言之，类封装了一系列方法，并且可以通过一定的规则约定方法访问权限。在 Python 中没有 public、protected、private 之类的访问权限控制修饰词，Python 通过方法名约定访问权限。

例如：

（1）普通名字，表示 public。

（2）以_前导的名字，从语法上视为 public，但约定俗称的意思是“可以被访问，但请视为 private，不要随意访问”。

（3）以__前导、以__后缀的名字，特殊属性，表示 public。

（4）以__前导、不以__后缀的名字，表示 private。

private 名字不能被继承类引用。private 不允许通过实例对象直接访问，本质上是因为 private 属性名被 Python 解释器改成类名属性名了，因此仍然可以通过类名属性名访问 private 属性，但是不同版本的 Python 解释器改造的规则不一致，通常不建议用户这样访问 private 属性，因为代码不具有可移植性。

## 4.2 认识 Python 中的类、对象和方法

### 4.2.1 类的定义与创建


视频讲解

类（Class）可以看作是类别或者种类的同义词。在 Python 中类用来描述具有相同属性和方法的对象的集合，它定义了该集合中每个对象所共有的属性和方法。使用类几乎可以模拟任何东西。

例如，设想自己正走在大街上，从身前跑过一只猫，猫是“猫类”的实例，这就是一个有很多子类的一般类，从自己身前跑过的猫可能属于子类“波斯猫类”。这里可以将“猫类”想象成是所有猫的集合，而“波斯猫类”是其中的一个子集。当一个对象所属的类是另一个对象所属类的子集时就称前者为后者的子类（Subclass）。所以，“波斯猫类”是“猫类”的子类；相反，“猫类”是“波斯猫类”的超类（Superclass）。

这里的猫是指从自己身前跑过的那只特定的猫，但“猫类”或者“波斯猫类”表示的不是特定的猫，而是任何猫或者任何波斯猫。对于大多数宠物猫，大家都了解些什么呢？这些猫都有年龄和名字，当然猫还会打滚和下蹲。因为猫都具有上述两项信

息(年龄和名字)和两种行为(打滚和下蹲),依据这些,就可以创建一个用来表示猫的简单类 Cat。

下面使用 Python 中的类定义语言来创建简单类 Cat,并借此讲解这个类定义函数。类是一个用户定义类型,和其他大多数计算机语言一样,Python 使用关键字 class 来定义类。函数的语法格式如下:

```
class classname:
<statement-1>
…
<statement-n>
```

<statement-1>与<statement-n>之间可以包含任何有效的 Python 语句,用来定义类的属性与方法。下面创建简单类 Cat。

**【例 4-2】** Cat 类的定义。

```
class Cat():
    #一次模拟小猫的简单尝试

    def __init__(self, name, age):
        #初始化属性 name 和 age
        self.name = name
        self.age = age

    def sit(self):
        #模拟小猫被命令下蹲
        print(self.name.title() + " is now sitting.")
    def roll_over(self):
        #模拟小猫被命令打滚
        print(self.name.title() + " rolled over!")
```

通过以上代码就可以创建简单类 Cat,赋予每只小猫下蹲(sit())和打滚(roll_over())的行为。

在这段代码中,用 class 定义了一个名为 Cat 的类。根据约定,在 Python 中首字母大写的名称指的是类。在本代码的第 1 行,这个类定义中的括号是空的,这是因为要创建一个空白类。在第 2 行编写了一个文档字符串,对这个类的功能进行了描述。从第 4 行开始,在类中定义了 3 个函数,类中的函数称为方法。前面学过的有关函数的一切都适用于方法,唯一重要的差别是调用方法的方式不同。第 1 个方法__init__()是特殊的方法,每当用户根据 Cat 类创建新实例时 Python 都会自动运行它。这里将

__init__()方法定义成包含 3 个形参，即 self、name、age，形参 self 必不可少，还必须位于其他形参的前面。那么为什么必须在方法定义中包含形参 self 呢？这是因为 Python 在调用这个__init__()方法创建 Cat 实例时将自动传入实参 self。每个与类相关联的方法调用都自动传递实参 self，它是一个指向实例本身的引用，让实例能够访问类中的属性和方法。在创建 Cat 实例时，Python 将调用 Cat 类的方法__init__()。程序将通过实参向 Cat()传递名字和年龄；self 会自动传递，因此不需要手动去传递它。每当根据 Cat 类创建实例时都只需要给最后两个形参(name 和 age)提供值。

第 6、7 行定义的两个变量都有前缀 self。以 self 为前缀的变量可供类中的所有方法使用，还可以通过类的任何实例来访问这些变量。self. name＝name 获取存储在形参 name 中的值，并将其存储到变量 name 中，然后该变量被关联到当前创建的实例。self. age＝age 的作用与此类似。像这样可通过实例访问的变量称为属性。

Cat 类还定义了另外两个方法，即 sit()和 roll_over()。由于这些方法不需要额外的信息，例如名字或年龄，所以它们只有一个形参 self。在后面将创建的实例都能够访问这些方法，换句话说，它们都会下蹲和打滚。当前，sit()和 roll_over()所做的有限，它们只是打印一条消息，指出小猫正下蹲或打滚。用户可以扩展这些方法来模拟实际情况：如果这个类包含在一个计算机游戏中，这些方法将包含创建小猫下蹲和打滚动画效果的代码。如果这个类是用于控制机器猫的，这些方法将引导机器猫做出下蹲和打滚的动作。

接下来创建一个表示特定小猫的实例。首先可以将类视作有关如何创建实例的说明，即可以理解成 Cat 类是一系列说明，让 Python 知道如何创建一个表示特定小猫的实例。

**【例 4-3】** 小猫实例。

```
my_cat = Cat("tommy",3)

print("my cat's name is " + my_cat.name.title() + ".")
print("my cat is " + str(my_cat.age) + "years old.")
```

这里使用的是前一个示例中编写的 Cat 类。在第 1 行代码处，让 Python 创建一只名字为 tommy、年龄为 3 的小猫。当遇到这行代码时，Python 使用实参"tommy"和 3 调用 Cat 类中的方法__init__()。方法__init__()创建一个表示特定小猫的实例，并使用外部提供的值来设置属性 name 和 age。方法__init__()并未显式地包含

return 语句,但 Python 自动返回一个表示这只小猫的实例,然后将这个实例存储在变量 my_cat 中。在这里命名约定很有用:通常可以认为首字母大写的名称(例如 Cat)指的是类,而小写的名称(例如 my_cat)指的是根据类创建的实例。

在本节前面创建了一个简单的 Cat 类,并在方法__init__()中定义了 name 和 age 属性。如果要访问实例的属性,可以使用句点表示法。在第 3 行编写了如下代码来访问 my_cat 的 name 属性的值:

```
my_cat.name
```

句点表示法在 Python 中很有用,这种语法演示了 Python 语句如何来获得属性的值。例如在这个示例中,Python 先找到 my_cat 实例,再查找与这个实例相关联的 name 属性。在 Cat 类中引用这个属性的时候使用的是 self.name。在代码的第 4 行,使用同样的方法来获取 age 属性的值。在代码的第 1 个 print()语句中,my_cat.name.title()将 my_cat 的 name 属性值"tommy"的首字母改成大写的'T';在第 2 个 print()语句中,str(my_cat.age)将 my_cat 的 age 属性值 3 转换成字符串类型。对于上述示例,my_cat 实例的输出结果如下:

```
my cat's name is Tommy.
my cat is 3years old.
```

### 4.2.2 构造函数

在 4.1.1 的示例代码中已经使用到类中的一个特殊方法——__init__(self,...),这个方法被称为构造函数,用来初始化对象(实例),在创建新对象时调用。__init__()方法在类的一个对象(实例)被建立时马上运行。这个方法可以用来对用户的对象做一些用户希望的初始化。构造函数属于每个对象,每个对象都有自己的构造函数。如果用户未设计构造函数,Python 将提供一个默认的构造函数。注意,这个名称的开始和结尾都是双下画线。构造函数的作用有两个:一是在内存中为类创建一个对象;二是调用类的初始化方法来初始化对象。

**【例 4-4】** 类的创建和实例化。

```
class Person:
    def __init__(self,name):
```

```
        self.name = name
    def sayHi(self):
        print ("Hello,my name is",self.name)
p = Person("python")
p. sayHi()
```

运行结果：

```
Hello, my name is python
```

__init__()方法定义为取一个参数 name(以及普通的参数 self)。在这个__init__()里只是创建一个新的域，也称为 name。注意它们是两个不同的变量，尽管它们有相同的名字。点号使用户能够区分它们。最重要的是，没有专门调用__init__()方法，只是在创建一个类的新实例的时候把参数包括在圆括号内跟在类名后面，从而传递给__init__()方法。这是这种方法的重要之处。

## 4.3 类的属性

### 4.3.1 类属性和实例属性

类中的属性分为两种，一是类属性，二是实例属性。类属性是在类中方法之外定义的；实例属性则是在构造函数__init__()中定义的，在定义时以 self 为前缀，只能通过对象名访问，上一节创建的 Cat 类中的 self. name 和 self. age 就是实例属性。为了更好地讲解类属性，这里将通过对上一节中创建的简单类 Cat 的属性进行增加和修改来说明。类属性的修改和增加都是通过“类名. 属性名”的方式直接进行的。

下面创建一个简单的类 Cat。

**【例 4-5】** 增加类属性。

```
class Cat():
#一次模拟小猫的简单尝试

#增加类属性
    Reproduction_way = "taisheng"
    Song_way = "miaomiao"
```

```
    def __init__(self, name, age):
    #初始化 name 和 age 属性
        self.name = name
        self.age = age

    def _sit_(self):
    #模拟小猫被命令下蹲
        print(self.name.title() + " is now sitting.")

    def roll_over(self):
    #模拟小猫被命令打滚
        print(self.name.title() + " rolled over!")
```

上面的代码增加了两个类属性，分别是生育后代的方式"taisheng"以及叫声"miaomiao"，这是猫类的共同属性。

### 4.3.2 公有属性和私有属性

在默认情况下，程序是可以从外部访问一个对象的特性的。有些程序员认为这样做是可以的，有些程序员(比如 SmallTalk 之父，SmallTalk 的对象特性只允许由同一个对象的方法访问)认为这样做是不可以的，觉得这样做就破坏了封装的原则。他们认为对象的状态对于外部应该是完全隐藏的(不可访问)。可能会有人感到奇怪，为什么他们会站在如此极端的立场上，每个对象管理自己的特性还不够吗？为什么还要对外部世界隐藏呢？毕竟如果能直接使用将会更方便。

关键在于其他程序员可能不知道(可能也不应该知道)用户的对象内部的具体操作。Python 并不直接支持私有方式，而是靠程序员自己把握在外部进行特性修改的时机。毕竟在使用对象前应该知道如何使用，但是可以用一些小技巧达到私有特性的效果。

如果想要让方法或者特性变为私有的，即从外部无法进行访问，只需要在它的名字前面加上双下画线即可。具体来讲，以__(双下画线)开头的属性是私有属性，否则这个属性就是公有属性。私有属性通过"对象名.类名__私有成员名"进行访问，不能在类外进行直接访问。举例如下：

**【例 4-6】** 定义私有属性。

```
class Secret():
    def __unaccessible(self):
```

```
        print ("Sorry , you can not accessible...")
    def  accessible(self):
        print ("Yes , you can accessible ,and the secret is ... ")
        self.__unaccessible()
```

现在,正如在这个例子中展示的,__unaccessible()是无法从外界进行访问的,但是从类的内部还是能够进行访问的(比如从 accessible()进行访问):

```
>>> s = Secret()
>>> s.__unaccessible()
```

运行结果:

```
Traceback (most recent call last):
  File"<pyshell#112>", line 1, in ?
    s.__unaccessible()
AttributeError: Secretive instance has no attribute '__unaccessible'
>>> s.accessible()
```

运行结果:

```
Yes , you can accessible ,and the secret is ...
Sorry , you can not accessible ...
```

双下画线虽然有些奇怪,但看起来像是其他编程语言中的标准的私有方法。事实上真正发生的事情是不标准的。因为在类的内部定义中,所有以双下画线开始的命名都将会被翻译成前面加单下画线和类名的形式。例如:

```
>>> Secret._Secret__unaccessible
```

运行结果:

```
<unbound method Secret.__unaccessible>
```

总体来说,想要确保其他人不会访问对象的方法和特性是不可能的,但是像这类的“名称变化术”就是他们不应该访问这些方法或者特性的强信号。

如果不想使用这种方法,但是又想让其他对象不能访问内部数据,那么可以使用双下画线。虽然这不过是一个习惯,但的确有实际效果。例如,前面有下画线的名字

都不会被带星号的 import 语句“form module import * ”导入。

有些编程语言支持多种层次的成员变量或特性私有化，比如在 Java 中就支持 4 种级别。尽管单、双下画线在某种程度上给出了两个级别的私有性，但是 Python 并没有真正的私有化支持。

## 4.4 类的方法

### 4.4.1 类方法的调用

在这里仍旧使用 4.2.1 小节中创建的 Cat 类实例，在创建 Cat 类实例后就可以使用句点表示法来调用 Cat 类中定义的任何方法。下面通过让小猫进行下蹲和打滚的动作来展示调用方法。

**【例 4-7】** 类方法调用的示例。

```
class Cat():
    == snip ==
my_cat = Cat("tommy",3)
my_cat._sit_()
my_cat.roll_over()
```

如果想要调用类中的方法，可以通过指定实例的名称（这里是 my_cat）和想要调用的方法，并用句点分隔它们。当遇到代码 my_cat._sit_()时，Python 在 Cat 类中查找sit()方法并运行其代码块。同样，Python 也会用一样的方式解读代码 my_cat.roll_over()。

### 4.4.2 类方法的分类


视频讲解

在类中可以根据需要定义一些方法，定义方法使用 def 关键字，在类中定义的方法至少会有一个参数，一般以名为“self”的变量作为该参数（用其他名称也可以），而且需要作为第一个参数。在 Python 中类的方法大致可以分为 3 类，即类方法、实例方法和静态方法。

类方法是类对象所拥有的方法，需要用修饰器“@classmethod”来标识其为类方法。它能够通过实例对象和类对象去访问。类方法的用途就是可以对类属性进行修改。对于类方法，第一个参数必须是类对象，一般以“cls”作为第一个参数。举例

如下：

【例 4-8】 类方法示例。

```
class people:
country = "china"
@classmethod
  def getCountry(cls):            #类方法
        return cls.country
```

实例方法是在类中最常定义的成员方法，它至少有一个参数，并且必须以实例对象作为其第一个参数，一般以名为“self”的变量作为第一个参数(注意，不能通过类对象引用实例方法)。举例如下：

【例 4-9】 实例方法示例。

```
@classmethod
    def setCountry(cls,country):
        cls.country = country
class InstanceMethod(object):
    def __init__(self, a):
        self.a = a
    def f1(self):
        print ("This is {0}".format(self))
    def f2(self, a):
        print ("Value:{0}".format(a))
```

静态方法需要通过修饰器“@staticmethod”来进行修饰，静态方法对参数没有要求，不需要多定义参数。在静态方法中只能访问属于类的成员，不能访问属于对象的成员，而静态方法也只能通过类名调用。举例如下：

【例 4-10】 静态方法示例。

```
country = "china"
@staticmethod
  def getcountry():
  return people.country
@staticmethod
  def setcountry(countryName):
    people.country = countryName
```

对于这 3 种不同的方法出现了一个问题：既然有了实例方法，类方法和静态方法与之相比又有什么好处呢？具体地讲，在类方法中不管是使用实例还是类调用方法，

都会把类作为第一个参数传递进来，这个参数就是类本身。如果继承了这个使用类方法的类，则该类的所有子类都会拥有这个方法，并且这个方法会自动指向子类本身。静态方法是和类与实例都没有关系的，完全可以使用一般方法代替，但是使用静态方法可以更好地组织代码，防止代码较多时变得比较混乱。类方法是可以代替静态方法的。静态方法不能在继承中修改。

### 4.4.3　析构函数

在 Python 中没有专用的构造和析构函数，但是一般可以用__init__()和__del__()分别完成初始化和删除操作，因此可用它们代替构造和析构函数。从这个意义上讲，__init__()方法属于 Python 语言的构造函数；__del__()方法属于 Python 语言的析构函数，它在对象消逝的时候被调用，用来释放对象占用的资源。析构函数在对象就要被垃圾回收之前调用，但发生调用的具体时间是不可知的。这里通过一个例子来说明 Python 的析构函数，举例如下：

**【例 4-11】** 析构函数示例。

```
class test():
    def __init__(self):
        print("AAA")
    def __del__(self):
        print("BBB")
    def my(self):
        print("CCC")
>>> obj = test()
AAA
BBB
>>> obj.my()
CCC
>>> del obj
BBB
```

上述例子中的__del__()函数就是一个析构函数了，当使用 del 删除对象时会调用它本身的析构函数。另外，当对象在某个作用域中调用完毕后，在跳出其作用域的同时析构函数也会被调用一次，这样可以用来释放内存空间。__del__()也是可选的，如果不提供，则 Python 会在后台提供默认析构函数。如果要显式地调用析构函数，可以使用 del 关键字，方式如下：

```
del 对象名
```

## 4.5 类的继承

代码重用是软件工程的重要目标之一，类的重用是面向对象的核心内容之一，在编写类时并非总是要重新开始。如果用户要编写的类是另一个现成类的特殊版本，可以使用继承，在这个现成类的基础上创建新类，在所创建的新类中通过添加代码来扩展现成类的属性和方法，这样不仅能够减少工作量，而且能降低出现错误的可能性。

### 4.5.1 父类与子类

当一个类继承另一个类时，它将自动获得另一个类的所有属性和方法，原有的类称为基类、父类或超类(Baseclass、Superclass)，而新类称为子类(Subclass)。子类继承了其父类的所有属性和方法，同时还可以定义自己的属性和方法。

接下来给出更详细的定义，父类是指被直接或间接继承的类。在 Python 中 object 类是所有类的直接或间接父类。在继承关系中，继承者是被继承者的子类。子类继承所有祖先的非私有属性和非私有方法，子类也可以增加新的属性和方法，子类还可以通过重定义覆盖从父类中继承而来的方法。

### 4.5.2 继承的语法


视频讲解

对于继承的语法，下面通过一个实例来展示说明。

例如已经编写了一个名为 Animal 的 Class，有一个 run()方法可以直接打印。

**【例 4-12】** Animal 继承。

```
class Animal(object):
    def run(self):
        print("Animal is running...")
```

当需要编写 Dog 和 Cat 类时就可以直接从 Animal 类继承：

```
class Dog(Animal):
    pass
class Cat(Animal):
    pass
```

对于 Dog 来说,Animal 就是它的父类;对于 Animal 来说,Dog 就是它的子类。Cat 和 Dog 类似。在创建子类时,父类必须包含在当前文件中,且位于子类前面。这里定义了子类 Dog 和 Cat。在定义子类时,必须在括号内指定父类的名称。

那么继承有什么好处?最大的好处是子类获得了父类的全部功能。由于 Animal 实现了 run()方法,因此 Dog 和 Cat 作为它的子类什么事也没干就自动拥有了 run()方法:

```
dog = Dog()
dog.run()
cat = Cat()
cat.run()
```

运行结果如下:

```
Animal is running...
Animal is running...
```

继承的第二个好处需要用户对代码做一点改进。大家可以看到,无论是 Dog 还是 Cat,它们在 run()的时候显示的都是 Animal is running...,符合逻辑的做法是分别显示 Dog is running...和 Cat is running...,因此对 Dog 和 Cat 类改进如下:

```
class Dog(Animal):
    def run(self):
        print("Dog is running...")

class Cat(Animal):
    def run(self):
        print("Cat is running...")
```

再次运行,结果如下:

```
Dog is running...
Cat is running...
```

当子类和父类存在相同的 run()方法时，子类的 run()覆盖父类的 run()，在代码运行时总是会调用子类的 run()。

当然，用户也可以给子类增加一些方法，比如 Dog 类：

```
class Dog(Animal):
    def run(self):
        print("Dog is running...")

    def eat(self):
        print("Eating meat...")
```

### 4.5.3 多重继承

继承是面向对象编程的一个重要方式，因为通过继承，子类就可以扩展父类的功能。

这里想一下 Animal 类层次的设计，假设要实现 Dog(狗)、Bat(蝙蝠)、Parrot(鹦鹉)、Ostrich(鸵鸟)4 种动物，如果按照哺乳动物和鸟类归类，可以设计出如下类层次。

- 哺乳类：能跑的哺乳类，能飞的哺乳类。
- 鸟类：能跑的鸟类，能飞的鸟类。

如果要再增加“宠物类”和“非宠物类”，那么类的数量会呈指数增长，很明显这样设计是不行的，正确的做法是采用多重继承。首先，主要的类层次仍按照哺乳类和鸟类设计。

**【例 4-13】** 多重继承示例。

```
class Animal(object):
    pass
#大类
class Mammal(Animal):
    pass
class Bird(Animal):
    pass
#各种动物
class Dog(Mammal):
    pass
class Bat(Mammal):
    pass
```

```
class Parrot(Bird):
    pass
class Ostrich(Bird):
    pass
```

现在要给动物加上 Runnable 和 Flyable 的功能，只需要先定义好 Runnable 和 Flyable 的类即可：

```
class Runnable(object):
    def run(self):
        print("Running...")
class Flyable(object):
    def fly(self):
        print("Flying...")
```

对于需要 Runnable 功能的动物，就多继承一个 Runnable，例如 Dog：

```
class Dog(Mammal, Runnable):
    pass
```

对于需要 Flyable 功能的动物，就多继承一个 Flyable，例如 Bat：

```
class Bat(Mammal, Flyable):
    pass
```

通过多重继承，一个子类就可以同时获得多个父类的所有功能。

在设计类的继承关系时，通常主线都是单一继承下来的，例如 Ostrich 继承自 Bird。但是，如果需要"混入"额外的功能，通过多重继承就可以实现，比如让 Ostrich 除了继承自 Bird 外，再同时继承 Runnable。这种设计通常称为 Mixin。

为了更好地看出继承关系，把 Runnable 和 Flyable 改为 RunnableMixin 和 FlyableMixin。类似地，用户还可以定义出肉食动物 CarnivorousMixin 和植食动物 HerbivoresMixin，让某个动物同时拥有几个 Mixin：

```
class Dog(Mammal,RunnableMixin, CarnivorousMixin):
    pass
```

Mixin 的目的就是给一个类增加多个功能，这样在设计类的时候可以优先考虑通过多重继承来组合多个 Mixin 的功能，而不是设计多层次的复杂的继承关系。

Python 自带的很多库也使用了 Mixin。举个例子，Python 自带了 TCPServer 和 UDPServer 这两类网络服务，若要同时服务，多个用户就必须使用多进程或多线程模型，这两种模型由 ForkingMixin 和 ThreadingMixin 提供。通过组合，用户就可以创造出合适的服务。

这样，用户不需要复杂而庞大的继承链，只要选择组合不同的类的功能，就可以快速构造出所需的子类。

### 4.5.4 运算符的重载

在 Python 类中可以重写某些运算符的方法函数，例如类中提供了__add__()这个钩子函数，当调用"＋"(加法)运算时，实际上是调用了__add__()钩子函数，用户在类中可以重写这些钩子函数。

在 Python 中带有前/后缀、双下画线的方法函数称为钩子函数，钩子函数具有以下特征：

(1) 多数钩子函数均可在类中被重写。

(2) 钩子函数无预设值。

(3) 相应运算符调用时会自动映射调用这些钩子函数。

表 4.1 列举了一些常见的运算符重载方法。

**表 4.1 常见的运算符重载方法**

| method | overload | call |
|---|---|---|
| __init__() | 构造函数 | 对象创建：X=Class(args) |
| __del__() | 析构函数 | X 对象收回 |
| __add__() | 运算符＋ | X+Y、X+=Y |
| __or__() | 运算符\| | X\|Y、X\|=Y |
| __repr__()、__str__() | 打印、转换 | print(X)、repr(X)、str(X) |
| __call__() | 函数调用 | X(*args, **kwargs) |
| __getattr__() | 点号运算 | X.undefined |
| __setattr__() | 属性赋值 | X.any=value |
| __delattr__() | 属性删除 | delX.any |
| __getattribute__() | 属性获取 | X.any |
| __getitem__() | 索引运算 | X[key], X[i:j] |
| __setitem__() | 索引赋值 | X[key], X[i:j]=sequence |
| __delitem__() | 索引和分片删除 | del X[key], del X[i:j] |
| __len__() | 长度 | len(X) |
| __bool__() | 布尔测试 | bool(X) |

续表

| method | overload | call |
| --- | --- | --- |
| __lt__()、__gt__()、__le__()、__ge__()、__eq__()、__ne__() | 特定的比较 | X<Y、X>Y、X<=Y、X>=Y、X==Y、X!=Y |
| __radd__() | 右侧加法 | other+X |
| __iadd__() | 实地(增强的)加法 | X+=Y(或__add__()) |
| __iter__()、__next__() | 迭代环境 | I=iter(X)，next() |
| __contains__() | 成员关系测试 | item in X(任何可迭代) |
| __index__() | 整数值 | hex(X)、bin(X)、oct(X) |
| __enter__()、__exit__() | 环境管理器 | withobj as var: |
| __get__()、__set__()、__delete__() | 描述符属性 | X. attr，X. attr=value，del X. attr |
| __new__() | 创建 | 在__init__()之前创建对象 |

## 4.6　类的组合

前面讲了面向对象与类的继承，大家知道了继承是一种“什么是什么”的关系。然而类与类之间还有另一种关系，那就是组合，这是类的另一种重用方式。如果程序中的类需要使用一个其他对象，就可以使用类的组合方式。在 Python 中，一个类可以包含其他类的对象作为属性，这就是类的组合。

下面看一个例子，先定义一个老师类，老师类有名字、年龄、出生的年/月/日、所教的课程等特征以及走路和教书的技能。

**【例 4-14】**　类组合示例。

```
class Teacher:
    def __init__(self,name,age,year,mon,day):
        self.name = name
        self.age = age
        self.year = year
        self.mon = mon
        self.day = day

    def walk(self):
        print("%s is walking slowly" % self.name)

    def teach(self):
        print("%s is teaching" % self.name)
```

再定义一个学生类，学生类有名字、年龄、出生的年/月/日、学习的组名等特征以及走路和学习的技能：

```
class Student:
    def __init__(self,name,age,year,mon,day):
        self.name = name
        self.age = age
        self.year = year
        self.mon = mon
        self.day = day

    def walk(self):
        print("%s is walking slowly" % self.name)

    def study(self):
        print("%s is studying" % self.name)
```

根据类的继承这个特性，可以把代码缩减一下。首先定义一个人类，然后让老师类和学生类继承人类的特征和技能：

```
class People:
    def __init__(self,name,age,year,mon,day):
        self.name = name
        self.age = age
        self.year = year
        self.mon = mon
        self.day = day

    def walk(self):
        print("%s is walking" % self.name)
class Teacher(People):
    def __init__(self,name,age,year,mon,day,course):
        People.__init__(self,name,age,year,mon,day)
        self.course = course

    def teach(self):
        print("%s is teaching" % self.name)
class Student(People):
    def __init__(self,name,age,year,mon,day,group):
        People.__init__(self,name,age,year,mon,day)
        self.group = group

    def study(self):
        print("%s is studying" % self.name)
```

再对老师和学生进行实例化，得到一个老师和一个学生：

```
t1 = Teacher("alex",28,1989,9,2,"python")
s1 = Student("jack",22,1995,2,8,"group2")
```

现在想知道 t1 和 s1 的名字、年龄、出生的年/月/日都很容易，但是想一次性打印出 t1 或 s1 的生日就不那么容易了，这时需要用字符串进行拼接，有没有什么更好的办法呢？

有，那就是组合。

继承是一个子类与一个父类的关系，而组合是一个类与另一个类的关系。

可以说每个人都有生日，而不能说人是生日，这样就要使用组合的功能。

可以把出生的年/月/日再另外定义一个日期的类，然后用老师或者学生类与这个日期的类组合起来，就可以很容易地得出老师 t1 或者学生 s1 的生日，再也不用字符串拼接那么麻烦。请看下面的代码：

```
class Date:
    def __init__(self,year,mon,day):
        self.year = year
        self.mon = mon
        self.day = day

    def birth_info(self):
        print("The birth is %s-%s-%s" % (self.year,self.mon,self.day))

class People:
    def __init__(self,name,age,year,mon,day):
        self.name = name
        self.age = age
        self.birth = Date(year,mon,day)

    def walk(self):
        print("%s is walking" % self.name)

class Teacher(People):
    def __init__(self,name,age,year,mon,day,course):
        People.__init__(self,name,age,year,mon,day)
        self.course = course

    def teach(self):
        print("%s is teaching" % self.name)

class Student(People):
```

```
    def __init__(self,name,age,year,mon,day,group):
        People.__init__(self,name,age,year,mon,day)
        self.group = group

    def study(self):
        print("%s is studying" % self.name)
t1 = Teacher("alex",28,1989,9,2,"python")
s1 = Student("jack",22,1995,2,8,"group2")
```

这样一来,就可以使用跟前面一样的方法来调用老师 t1 或学生 s1 的姓名、年龄等特征以及走路、教书或者学习的技能:

```
print(t1.name)
t1.walk()
t1.teach()
```

输出为:

```
alex
alex is walking
alex is teaching
```

那么怎么能够知道他们的生日呢? 可以如下:

```
print(t1.birth)
```

输出为:

```
<__main__.Date object at 0x0000000002969550>
```

这个 birth 是子类 Teacher 从父类 People 继承过来的,而父类 People 的 birth 又是与 Date 类组合在一起的,所以这个 birth 是一个对象。在 Date 类下面有一个 birth_info 的技能,这样就可以通过调用 Date 下面的 birth_info()这个函数属性来知道老师 t1 的生日了:

```
t1.birth.birth_info()
```

得到的结果为:

```
The birth is 1989 - 9 - 2
```

如果想知道学生 s1 的生日，使用同样的方法：

```
s1.birth.birth_info()
```

得到的结果为：

```
The birth is 1995 - 2 - 8
```

组合就是在一个类中使用到另一个类，从而把几个类拼到一起。组合是为了减少重复代码。

在实际的项目开发过程中，如果只使用继承和组合中的一种技术，是很难满足实际需求的，所以在实际的开发过程中开发人员通常会将两种技术结合起来使用。

## 4.7　类的异常处理

异常处理在任何一门编程语言里都是被关注的一个话题，良好的异常处理可以让程序更加健壮，清晰的错误信息更能帮助程序开发人员快速修复问题。在 Python 中，和部分高级语言一样，使用了 try/except 语句块来处理异常，如果用户有其他编程语言的经验，实践起来并不难。

### 4.7.1　异常

异常即是在程序执行过程中发生的影响程序正常运行的一个事件，该事件会在程序执行过程中发生，影响了程序的正常执行。一般情况下，在 Python 无法正常处理程序时就会发生一个异常。异常是 Python 对象，表示一个错误。当 Python 脚本发生异常时我们需要捕获处理它，否则程序会终止执行。异常处理使程序能够处理完异常后继续它的正常执行，不至于使程序因异常导致退出或崩溃。

举一个具体的例子：打开一个不存在的文件。代码如下：

**【例 4-15】** 异常举例。

```
fr = open("/not there","r")
```

运行结果：

```
Traceback (most recent call last):
  File "tiaoshi005.py", line 1, in <module>
    fr = open("/notthere","r")
FileNotFoundError: [Errno 2] No such file or directory: '/not there'
```

例子中的代码视图打开一个不存在的文件，运行之后，抛出 FileNotFoundError 异常。

## 4.7.2 Python 中的异常类

Python 程序出现异常时将抛出一个异常类的现象。Python 中所有的异常类的根类都是 BaseException 类，它们都是 BaseException 的直接或间接子类。大部分常规异常类的基类是 Exception 的子类。

不管程序是否正常退出，都将引发 SystemExit 异常。例如，在代码中的某个位置调用了 sys.exit()函数时将触发 SystemExit 异常。利用这个异常，可以阻止程序退出或让用户确认是否真的需要退出程序。

表 4.2 列出了 Python 中内置的标准异常类，自定义异常类都是继承自这些标准异常类。

**表 4.2 Python 内置的标准异常类**

| 异常名称 | 描　述 |
|---|---|
| BaseException | 所有异常的基类 |
| SystemExit | 解释器请求退出 |
| KeyboardInterrupt | 用户中断执行(通常是输入^C) |
| Exception | 常规错误的基类 |
| StopIteration | 迭代器没有更多的值 |
| GeneratorExit | 生成器(generator)发生异常来通知退出 |
| StandardError | 所有的内建标准异常的基类 |
| ArithmeticError | 所有数值计算错误的基类 |
| FloatingPointError | 浮点计算错误 |
| OverflowError | 数值运算超出最大限制 |
| ZeroDivisionError | 除(或取模)零 (所有数据类型) |
| AssertionError | 断言语句失败 |
| AttributeError | 对象没有这个属性 |
| EOFError | 没有内建输入，到达 EOF 标记 |
| EnvironmentError | 操作系统错误的基类 |

续表

| 异常名称 | 描述 |
|---|---|
| IOError | 输入/输出操作失败 |
| OSError | 操作系统错误 |
| WindowsError | 系统调用失败 |
| ImportError | 导入模块/对象失败 |
| LookupError | 无效数据查询的基类 |
| IndexError | 序列中没有此索引(index) |
| KeyError | 映射中没有这个键 |
| MemoryError | 内存溢出错误(对于 Python 解释器不是致命的) |
| NameError | 未声明/初始化对象 (没有属性) |
| UnboundLocalError | 访问未初始化的本地变量 |
| ReferenceError | 弱引用(Weak reference)试图访问已经垃圾回收了的对象 |
| RuntimeError | 一般的运行时错误 |
| NotImplementedError | 尚未实现的方法 |
| SyntaxError | Python 语法错误 |
| IndentationError | 缩进错误 |
| TabError | Tab 和空格混用 |
| SystemError | 一般的解释器系统错误 |
| TypeError | 对类型无效的操作 |
| ValueError | 传入无效的参数 |
| UnicodeError | Unicode 相关的错误 |
| UnicodeDecodeError | Unicode 解码时错误 |
| UnicodeEncodeError | Unicode 编码时错误 |
| UnicodeTranslateError | Unicode 转换时错误 |
| Warning | 警告的基类 |
| DeprecationWarning | 关于被弃用的特征的警告 |
| FutureWarning | 关于构造将来语义会有改变的警告 |
| OverflowWarning | 旧的关于自动提升为长整型(long)的警告 |
| PendingDeprecationWarning | 关于特性将会被废弃的警告 |
| RuntimeWarning | 可疑的运行时行为(runtime behavior)的警告 |
| SyntaxWarning | 可疑的语法的警告 |
| UserWarning | 用户代码生成的警告 |

### 4.7.3 捕获与处理异常

捕捉异常通常使用 try/except 语句。

try/except 语句用来检测 try 语句块中的错误,从而让 except 语句捕获异常信息并处理。如果用户不想在异常发生时结束程序,只需在 try 里捕获它。

以下为简单的 try...except 的语法：

```
try:
<语句>                #运行别的代码
except <名字>:
<语句>                #如果在 try 部分引发了'name'异常
except <名字>,<数据>:
<语句>                #如果引发了'name'异常，获得附加的数据
```

当开始一个 try 语句后，Python 就在当前程序的上下文中作标记，这样当异常出现时就可以回到这里，try 子句先执行，接下来会发生什么依赖于执行时是否出现异常。

如果当 try 后的语句执行时发生异常，Python 就跳回到 try 并执行第一个匹配该异常的 except 子句，异常处理完毕，控制流就通过整个 try 语句（除非在处理异常时又引发新的异常）。

如果在 try 后的语句里发生了异常，却没有匹配的 except 子句，异常将被递交到上层的 try，或者到程序的最上层（这样将结束程序，并打印默认的出错信息）。

如果在 try 子句执行时没有发生异常，Python 将执行 else 语句后的语句（如果有 else），然后控制流通过整个 try 语句。

当然也可以不带任何异常类型使用 except，示例如下：

```
try:
    正常的操作
  ...
except:
    发生异常，执行这块代码
  ...
else:
    如果没有异常执行这块代码
```

以上方式 try-except 语句捕获所有发生的异常。但这不是一个很好的方式，我们不能通过该程序识别出具体的异常信息，因为它捕获所有的异常。

接下来，将结合 4.7.1 节中的例子具体讲述一下 try-except 语句的使用方法。在 4.7.1 节中举了一个打开不存在文件，然后抛出 FileNotFoundError 异常的例子，下面就使用 try-except 语句来进行异常捕获和处理。

【例 4-16】　try-except 语句的使用。

```
try:
    fr = open("/notthere")
exceptFileNotFoundError:
    print("This file is not exist!")
```

运行结果：

```
This file is not exist!
```

同时，再利用一个例子来阐述 try-except-else 语句的使用方法，具体如下：

【例 4-17】　try-except-else 语句的使用。

```
a = 1
b = 2
c = "1"
try :
    assert a < b
    d = a  +  b
except AssertionError as e:
    print ( "a < b" )
except TypeError as e:
    print (e)
else :
    print ( "Program execution successful" )
```

运行结果如下：

```
Program execution successful
```

在这个代码块中，尝试捕获处理两个异常，分别是 AssertionError 异常和 TypeError 异常。但是，程序运行顺利，没有发生异常，所以执行 else 语句。

下面将通过例子的讲述对异常处理做进一步的讲解。

该案例是一个猜数字游戏，先看一下最简单的猜数字的游戏，随机取 1～10，然后让游戏者猜。代码如下：

【例 4-18】　猜数字游戏。

```
import random
num = random.randint(1,10)
```

```
while True:
    guess = int(raw_input('Enter 1～10'))
    if guess > num:
        print("guess Bigger : ",guess)
    elif guess < num:
        print("guess Smaller : ",guess)
    else:
        print("Great,You guess correct.game over !")
        Break
```

运行之后的结果如下：

```
>>> Enter 1～10:5
guess Bigger: 5
>>> Enter 1～10:3
guess Bigger: 3
>>> Enter 1～10:2
guess Bigger: 2
>>> Enter 1～10:1
Great,You guess correct.Game Over
```

这个是没有异常保护的，若正常输入没有问题，但是若恶意输入 abc 或者是非数字，那就会有问题了：

```
>>> Enter 1～10:aa
ValueError: invalid literal for int() with base 10: 'abc'
```

所以要加入异常处理，具体如下：

```
import random
num = random.randint(1,10)
while True:
try:
        guess = int(raw_input('Enter 1～10'))
    except Exception,e:
        print("Input error!Please enter num :1～10")
        continue
    if guess > num:
        print ("guess Bigger : ",guess)
    elif guess < num:
        print ("guess Smaller : ",guess)
else:
        print("Great,You guess correct.game over !")
        Break
```

在以上代码中加入了异常处理语句，这样在输入非数字的时候，程序就可以捕获该错误，保证程序的正常运行。

### 4.7.4　自定义异常类

通过创建一个新的异常类，程序可以命名自己的异常。自定义异常应该是通过直接或间接的方式继承自典型的 Exception 类。

以下为与 RuntimeError 相关的实例，在实例中创建了一个类，基类为 RuntimeError，用于在异常触发时输出更多的信息。在 try 语句块中，用户自定义的异常后执行 except 块语句，变量 e 是用于创建 Networkerror 类的实例。

**【例 4-19】**　自定义异常类举例。

```
classNetworkerror(RuntimeError):
        def __init__(self, arg):
            self.args = arg
```

在定义以上类后，可以触发该异常，如下所示：

```
try:
    raise Networkerror("Bad hostname")
except (Networkerror) as e:
    print (e.args)
```

但是，因为 Networkerror 是一个自定义类，所以需要使用 raise 来显式地抛出异常。

自定义异常的其他使用方法则与标准模块中的异常类的使用方法一致。下面将通过一个具体的例子来进行自定义异常使用的详细讲解。具体如下：

**【例 4-20】**　判断输入的长短。

```
class ShortInputException(Exception):
    #A user - defined exception class.
        def __init__(self, length, atleast):
            Exception.__init__(self)
            self.length = length
            self.atleast = atleast
```

```
try:
    s = raw_input('Enter something -->')
if len(s) < 3:
    raise ShortInputException(len(s), 3)
else:
    print (s)
except EOFError:
    print '\nWhy did you do an EOF on me?'
except ShortInputException as x:
    print ('ShortInputException:The input was length %d, \
        was expecting at least %d.'% (x.length, x.atleast))
else:
    print ('No exception was raised.')
```

在上述例子中,先自定义了一个名为 ShortInputException 的异常类,其用来判断用户输入的字符串长度是否满足要求。在本例中,其判断输入字符串的长度是否大于或等于 3 个字符,若不满足,则抛出该异常。

### 4.7.5 with 语句

有一些任务,可能事先需要设置,事后做清理工作。对于这种场景,Python 的 with 语句提供了一种非常方便的处理方式。一个很好的例子是文件处理,用户需要获取一个文件句柄,从文件中读取数据,然后关闭文件句柄。

Python 中的 with 语句用于对资源进行访问的场合,保证不管处理过程中是否发生错误或者异常都会执行规定的__exit__("清理")操作,释放被访问的资源,比如有文件读写后自动关闭、线程中锁的自动获取和释放等。

与 Python 中 with 语句有关的概念有上下文管理协议、上下文管理器、运行时上下文、上下文表达式、处理资源的代码段。

with 语句的语法格式如下:

```
withcontext_expression [as target(s)]:
    with - body
```

下面将通过一个例子(打开文件的方法:文件不存在,或者读取数据失败,没有容错信息)来说明 with 语句与 try-except 语句之间的区别。

**【例 4-21】** with 语句与 try-except 语句的比较。

```
file = open("test.txt")
data = file.read()
file.close()
```

这里有两个问题：一是可能忘记关闭文件句柄；二是文件读取数据发生异常，没有进行任何处理。下面是处理异常的加强版本：

```
try:
    file = open("test.txt")
    data = file.read()
    do something
finally:
    file.close()
```

虽然这段代码运行良好，但是太冗长了。这时候就是 with 一展身手的时候了。除了有更优雅的语法以外，with 还可以很好地处理上下文环境产生的异常。下面是 with 版本的代码：

```
with open("test.txt") as f:
    data = f.read()
    do something
```

在这个例子中，由于使用了 with 语句，不需要 try-finally 语句来确保文件对象的关闭。因为无论该程序是否会出现异常，文件对象都将被系统关闭。

但是并不是所有对象都支持 with 语句这一新特性的，只有支持上下文管理协议的对象才能使用 with 语句，支持该协议的对象有 file、decimal、Context、thread、LockType、threading.Lock、threading.RLock、threading.Condition、threading.Semaphore、threading.BoundedSemaphore。

### 4.7.6　断言

使用 assert 断言是学习 Python 的一个非常好的习惯，Python assert 断言语句的格式及用法很简单。在没完善一个程序之前，我们不知道程序在哪里会出错，与其让它在运行时崩溃，不如让它在出现错误条件时就崩溃，这时候就需要 assert 断言的帮助。

Python assert 断言的作用：Python assert 断言是声明其布尔值必须为真的判定，如果发生异常就说明表达式为假。可以理解 assert 断言语句为 raise-if-not，用来测试表达式，其返回值为假，就会触发异常。如果断言成功，则程序不会采取任何措施，否则就会触发 AssertionError 异常。

assert 断言语句的语法格式如下：

```
assert expression
assert 表达式
```

下面是一些有关 assert 用法的语句，供读者参考：

```
assert 1 == 1
assert 2 + 2 == 2 * 2
assert len(['my boy',12])< 10
assert range(4) == [0,1,2,3]
```

如何为 assert 断言语句添加异常参数？assert 的异常参数，其实就是在断言表达式后添加字符串信息，用来解释断言并更好地知道哪里出了问题。格式如下：

```
assert expression [, arguments]
assert 表达式 [, 参数]
```

下面将通过一个例子来展示 assert 的使用方法。具体如下：

**【例 4-22】** assert 断言的使用。

```
assert 4 == 3 + 1
assert 4 == 3 * 1
```

运行结果：

```
Traceback (most recent call last):
  File "tiaoshi006.py", line 2, in <module>
    assert 4 == 3 * 1
AssertionError
```

在这个例子中，第 1 行代码成功运行，但是第 2 行代码在运行过程中抛出了一个 AssertionError 异常。

接下来利用 try-except 语句来捕获处理 AssertionError 异常。具体如下：

```
try:
    assert 4 == 3 * 1
except AssertionError:
    print("The expression symbol is wrong!")
```

运行结果：

```
The expression symbol is wrong!
```

这样，通过使用 try-except 成功地捕获了断言失败异常。

## 本章小结

类是客观世界中事务的抽象，是一种广义的数据类型，根据类来创建对象被称为实例化，对象是类实例化后的变量。本章主要介绍面向对象编程类的定义、属性和方法，以及运算符的重载，还介绍了类的继承和组合这两种重用技术，最后以概念与实例相结合的方式详细讲述 Python 异常处理机制、内置异常类的类型、异常处理的语法结构、异常的检测和处理方法、自定义异常类的方法与使用。

## 习题

1. 创建一个名为 Fruit 的类，其中方法__init__()设置两个属性：fruit_name 和 fruit_price；并创建一个名为 describe_fruit()的方法和一个名为 number_fruit()的方法，前者打印出前述的两项信息，后者打印出一条消息，表明该水果的储备量充足。

根据这个类创建一个名为 fruit 的实例，分别打印其两个属性，再调用前述的两个方法。

2. 向题 1 编写完成的程序中添加一个名为 number_sell 的属性，并将其默认值设置为 0。根据这个类创建一个 fruit 实例，打印出有多少水果卖出了，然后修改这个值并再次打印它。

再向程序中添加一个名为 set_number_sell()的方法，它能让卖家设置每次出售的限量。调用这个方法并向它传递一个值，然后再次打印这个值。

3. 创建一个 BankAccount 类，表示银行账户。自行定义其中的属性和方法，并利用这个类创建一个账号名为 888666、余额为 35 000、年利率为 3%的银行账户，然后向其中存入 5000，取出 12 000。打印出账号、余额、年利率、年息。

4. 简要阐述继承和组合的概念以及两者的区别。

5. 自主设计一个程序案例，实现类的继承和组合。

6. 简要描述异常的处理机制。

7. 自行编程，利用 try-except 语句来实现捕获异常。

8. 自行编程，利用 with 语句和断言来实现检测和处理异常。

## 案例

在本章的最后，通过讲解一个文件读取的案例来展示一下处理多个异常的具体操作。在进行文件读取的处理时可能会遇到多个异常，接下来进行解释。

例如，当前目录下没有 test. txt 文件，然后执行下面的读取文件的代码：

**【例 4-23】** 多异常处理。

```
try:
    f = open("test.txt")
    line = f.read()
    num = int(line)
    print('"read num = %d" % num')
except IOError,e:
    print('"we catch IOError:",e')
finally:
    print("Close file")
    f.close()
```

在运行程序之后，就会捕获一个 IOError 错误，具体运行结果如下：

```
>>>
we catch IOError: [Errno 2] No such file or directory:'test.txt'
Close file
```

表明程序没有在当前目录下找到该文件，原本应该是返回一个异常，但是因为在程序中已经添加了异常处理语句，所以该异常被捕获并被处理，使得程序能正常运行。

接下来在当前目录下新建一个 test. txt 文件，并且在里面写上一个数字“50”，再

运行代码,就没有问题了,运行结果如下:

```
>>>
read num = 50
Close file
```

但是,若把 test.txt 文件里面的数字 50 改成字符串'abc',会出现什么情况呢?运行结果如下:

```
>>>
Close file
Traceback (most recent call last):
ValueError: invalid literal for int() with base 10: 'abc'
```

运行之后,就会报这是一个 ValueError,但是原先的代码只能捕捉 IOError,没有捕捉 ValueError,所以没有处理 except 部分,导致返回一个异常。继续修改代码,在原先的代码中加入 ValueError 异常处理,代码如下:

```
try:
    f = open("test.txt")
    line = f.read()
    num = int(line)
    print('"read num = %d" % num')
except IOError,e:
    print('"we catch IOError:",e')
except ValueError,e:
    print('"we catch ValueError:",e')
finally:
    print("Close file")
    f.close()
```

运行上述程序之后,结果如下:

```
>>>
we catch ValueError: invalid literal for int() with base 10: 'abc'
Close file
```

通过在程序中加入新的异常处理语句,就捕捉到了 ValueError,这说明 Python 在异常处理里面可以捕捉多个异常。也就是说,若发生了 IOError,就执行 IOError 里面的异常处理;若发生了 ValueError,就执行 ValueError 里面的异常。同样,还可以通过添加其他类型的异常处理语句对各种类型的异常进行处理。

# 第 5 章



# Python数据分析基础库

**本章学习目标：**

- 学习 NumPy 库的用法、数据结构和基本操作
- 学习 Pandas 库的用法、数据结构和基本操作
- 学习 Matplotlib 库的用法、数据结构和基本操作
- 掌握 SciPy 库的操作、作用
- 掌握 Scikit-learn 库的操作、作用

本章介绍 Python 进行数据分析时常用的 NumPy、Pandas、Matplotlib、SciPy 和 Scikit-learn 基础库。NumPy 是 Python 的一种开源数值计算扩展库，这种工具可用来存储和处理大型矩阵，比 Python 自身的嵌套列表(nested list structure)结构要高效许多；Pandas 是基于 NumPy 的一种工具，该工具是为了解决数据分析任务而创建的，Pandas 提供了大量的库和标准数据模型，以及高效、便捷地处理大型数据集所需的函数和方法；Matplotlib 是一个 Python 的 2D 绘图库，它基于各种硬拷贝格式和跨平台的交互式环境生成出版质量级别的图形；SciPy 是一款方便的专为科学和工程设计的 Python 工具包，包括统计、优化、整合、线性代数模块、傅里叶变换、信号和图像处理以及常微分方程求解器等；Scikit-learn(简称 Sklearn)是 SciPy 的扩展，建立在 NumPy 和 Matplotlib 库的基础之上，支持分类、回归、降维和聚类等机器学习算法。

## 5.1 NumPy

NumPy(Numerical Python 的缩写)是一个开源的 Python 科学计算库,包含很多实用的数学函数,涵盖线性代数运算、傅里叶变换和随机数生成等功能。NumPy 允许用户进行快速的交互式原型设计,可以很自然地使用数组和矩阵。它的部分功能如下。

(1) ndarray:一个具有矢量算术运算且节省空间的多维数组。

(2) 用于对整组数据进行快速运算的标准数学函数(无须编写循环)。

(3) 用于读/写磁盘数据的工具以及用于操作内存映射文件的工具。

(4) 线性代数、随机数生成以及傅里叶变换功能。

(5) 用于集成 C、C++、Fortran 等语言的代码编写工具。

NumPy 的底层算法在设计时就有着优异的性能,对于同样的数值计算任务,使用 NumPy 要比直接编写 Python 代码便捷得多。对于大型数组的运算,使用 NumPy 中数组的存储效率和输入/输出性能均优于 Python 中等价的基本数据结构(例如嵌套的 list 容器)。对于 TB 级的大文件,NumPy 使用内存映射文件来处理,以达到最优的数据读/写性能。这是因为 NumPy 能够直接对数组和矩阵进行操作,可以省略很多循环语句,其众多的数学函数也会让开发人员编写代码的工作轻松许多。不过 NumPy 数组的通用性不及 Python 提供的 list 容器,这是其不足之处。因此,在科学计算之外的领域,NumPy 的优势也就不那么明显了。NumPy 本身没有提供那么多高级的数据分析功能,理解 NumPy 数组以及面向数组的计算将有助于更加高效地使用诸如 Pandas 之类的工具。下面对 NumPy 的数据结构和操作进行介绍。

NumPy 的多维数组对象 ndarray 是一个快速、灵活的大数据集容器。用户可以利用这种数组对象对整块数据进行数学运算,其运算跟标量元素之间的运算一样。创建 ndarray 数组最简单的办法就是使用 array()函数。它接受一切序列型的对象(包括其他数组),然后产生一个新的、含有传入数据的 NumPy 数组。这里以一个列表的转换为例:

```
In [1]: import numpy as np
        data = [6, 7.5, 8, 0, 1]
        arr1 = np.array(data)
        arr1
Out[1]: array([ 6. ,  7.5,  8. ,  0. ,  1. ])
```

ndarray 是一个通用的同构数据多维容器，其中所有的元素必须是相同类型的。每一个数组都有一个 shape（表示维度大小的数组）和一个 dtype（用于说明数组数据类型的对象）：

```
In [2]: arr1.shape
Out[2]: (5,)
In [3]: arr1.dtype
Out[3]: dtype('float64')
```

嵌套序列（例如由一组等长列表组成的列表）将会被转换成一个多维数组：

```
In [4]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
        arr2 = np.array(data2)
        arr2
Out[4]:
array([[1, 2, 3, 4],
        [5, 6, 7, 8]])
In [5]: arr2.ndim
Out[5]: 2
In [6]: arr2.shape
Out[6]: (2, 4)
```

除非显式说明，否则 np.array()会尝试为新建的数组推断出一个较为合适的数据类型。数据类型保存在一个特殊的 dtype 对象中，例如上面的两个例子：

```
In [7]: arr1.dtype
Out[7]: dtype('float64')
In [8]: arr2.dtype
Out[8]: dtype('int32')
```

除了 np.array()外，还有一些函数可以新建数组，例如 np.zeros()和 np.ones()可以分别创建指定长度或形状的全为 0 或全为 1 的数组。Empty 可以创建一个没有任何具体数值的数组。如果要用这些方法创建数组，只需传入一个表示形状的元组即可：

```
In [9]: np.zeros(8)
Out[9]: array([ 0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.])
In [10]: np.zeros((2, 4))
Out[10]:
array([[ 0.,   0.,   0.,   0.],
        [ 0.,   0.,   0.,   0.]])
In [11]: np.empty((2, 3, 2))
Out[11]:
```

```
array([[[  9.78249979e-322,   0.00000000e+000],
        [  0.00000000e+000,   0.00000000e+000],
        [  0.00000000e+000,   0.00000000e+000]],

       [[  0.00000000e+000,   0.00000000e+000],
        [  0.00000000e+000,   0.00000000e+000],
        [  0.00000000e+000,   0.00000000e+000]]])
```

在 NumPy 中，np.empty()会认为返回全为 0 的数组是不安全的，所以它会返回一些未初始化的很接近 0 的随机值。

ndarray 的一些常用的基本数据操作函数如表 5.1 所示。

**表 5.1 ndarray 基本数据操作函数**

| 函　　数 | 说　　明 |
|---|---|
| array() | 将输入数据(列表、元组、数组或其他序列类型)转换为 ndarray。推断出 dtype 或特别指定 dtype，默认直接赋值输入数据 |
| asarray() | 将输入转化为 ndarray。如果输入本身是一个 ndarray，就不再复制 |
| arange() | 类似于内置的 range，但返回的是一个 ndarray 而非 list |
| ones()，ones_like() | 根据指定的形状和 dtype 创建一个全 1 数组。ones_like 以另一个数组为参数，并根据其形状和 dtype 创建一个全 1 数组 |
| zeros()，zeros_like() | 类似于 ones()和 ones_like()，只不过产生的是全 0 数组 |
| empty()，empty_like() | 创建新数组，只分配内存空间，不填充任何值 |
| full()，full_like() | 用 full value 中的所有值，根据指定的形状和 dtype 创建一个数组。full_like()使用另一个数组，用相同的形状和 dtype 创建 |
| eye()，identity() | 创建一个正方的 N×N 矩阵(对角线为 1，其余为 0) |

## 5.1.1 ndarray 的数据类型


视频讲解

dtype(数据类型)是一个特殊的对象，它含有 ndarray 将一块内存解释为特定数据类型所需的信息：

```
In [12]: arr3 = np.array([1, 2, 3], dtype = np.float64)
         arr3
Out[12]: array([ 1.,  2.,  3.])
In [13]: arr4 = np.array([1, 2, 3], dtype = np.int32)
         arr4
Out[13]: array([1, 2, 3])
```

dtype 是 NumPy 如此强大和灵活的原因之一。在多数情况下，它直接映射到相应的机器表示，这使得“读/写磁盘上的二进制数据流”以及“集成低级语言代码”等工作变得更加简单。数值型 dtype 的命名形式相同：一个类型名(例如 float 或 int)，后

面跟一个用于表示各元素位长的数字。标准的双精度浮点值(即 Python 中的 float 对象)需要占用 8 字节(即 64 位)。因此,该类型在 NumPy 中记作 float64。

可以用 astype 的方法显示更改数组的 dtype:

```
In [14]: arr5 = np.array([1, 2, 3])
  arr5.dtype
Out[14]: dtype('int32')
In [15]: arr6 = arr5.astype(np.float64)
  arr6
Out[15]: array([ 1.,  2.,  3.])
```

### 5.1.2 数组和标量之间的运算


视频讲解

用数组表达式代替循环的方法,通常被称作矢量化(vectorization)。大小相等的数组之间的任何算术运算都会应用到元素集:

```
In [16]: arr = np.array([[1. ,2. , 3.], [4. ,5. ,6]])
         arr * arr
Out[16]:
array([[  1.,   4.,   9.],
       [ 16.,  25.,  36.]])
In [17]: arr - arr
Out[17]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

同样,数组和标量的运算也会将那个标量传播到各个元素:

```
In [18]: 1 / arr
Out[18]:
array([[ 1.        ,  0.5       ,  0.33333333],
       [ 0.25      ,  0.2       ,  0.16666667]])
In [19]: arr * * 0.5
Out[19]:
array([[ 1.        ,  1.41421356,  1.73205081],
       [ 2.        ,  2.23606798,  2.44948974]])
```

### 5.1.3 索引和切片


视频讲解

NumPy 索引和切片是一个内容丰富的主题,因为选取数据子集或单个元素的方式有很多。首先,一维数组的切片索引基本和 Python 列

表的切片索引功能一致。

```
In [20]: arr = np.arange(10)
         arr
Out[20]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
In [21]: arr[4]
Out[21]: 4
In [22]: arr[3:7]
Out[22]: array([3, 4, 5, 6])
In [23]: arr[3:5] = 12
         arr
Out[23]: array([ 0,  1,  2, 12, 12,  5,  6,  7,  8,  9])
```

如上所示，当将一个标量赋值给一个切片时（例如 arr[3:5]=12），该值会自动传播到整个选区。因为数组切片是原始数组视图，这就意味着如果做任何修改，原始都会跟着更改。

```
In [24]: arr_slice = arr[3:5]
         arr_slice[1] = 100
         arr
Out[24]: array([  0,   1,   2,  12, 100,   5,   6,   7,   8,   9])
In [25]: arr_slice[:] = 250
         arr
Out[25]: array([  0,   1,   2, 250, 250,   5,   6,   7,   8,   9])
```

对于高维数组，能做的事情更多。在一个二维数组中，各索引位置上的元素不再是标量而是一维数组：

```
In [26]: arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
         arr[2]
Out[26]: array([7, 8, 9])
```

因此，可以对各个元素进行递归访问，但这样需要做的事情有点多。用户可以传入一个以逗号隔开的索引列表来选取单个元素。也就是说，下面这两种方式是等价的：

```
In [27]: arr[1][2]
Out[27]: 6
In [28]: arr[1, 2]
Out[28]: 6
```

花式索引是利用整数数组进行索引，假设有一个 8×4 的数组：

```
In [29]: arr = np.empty((8,4))
         for i in range(8):
             arr[i] = i
         arr
Out[29]:
array([[ 0.,   0.,   0.,   0.],
       [ 1.,   1.,   1.,   1.],
       [ 2.,   2.,   2.,   2.],
       ...,
       [ 5.,   5.,   5.,   5.],
       [ 6.,   6.,   6.,   6.],
       [ 7.,   7.,   7.,   7.]])
```

为了以特定的顺序选取行子集，只需传入一个用于指定顺序的整数列表或 ndarray 即可：

```
In [30]: arr[[4, 3, 0, 6]]
Out[30]:
array([[ 4.,   4.,   4.,   4.],
       [ 3.,   3.,   3.,   3.],
       [ 0.,   0.,   0.,   0.],
       [ 6.,   6.,   6.,   6.]])
```

使用负数索引将会从末尾开始选取行：

```
In [31]: arr[[-3, -5, -7]]
Out[31]:
array([[ 5.,   5.,   5.,   5.],
       [ 3.,   3.,   3.,   3.],
       [ 1.,   1.,   1.,   1.]])
```

当一次传入多个数组时，它返回的是一个一维数组，其中的元素对应各个索引元组：

```
In [32]: arr = np.arange(32).reshape((8,4))
         arr
Out[32]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       ...,
       [20, 21, 22, 23],
```

```
        [24, 25, 26, 27],
        [28, 29, 30, 31]])
In [33]: arr[[1,5,7,2], [0,3,1,2]]
Out[33]: array([ 4, 23, 29, 10])
```

它选出的元素其实是(1,0)、(5,3)、(7,1)和(2,2)这些位置的元素。这个花式索引的结果可能和某些用户预测的不太一样,选取矩阵的行列子集应该是矩形区域的形式才对。下面是得到该结果的一个办法:

```
In [34]: arr[[1,5,7,2]][:,[0,3,1,2]]
Out[34]:
array([[ 4,  7,  5,  6],
        [20, 23, 21, 22],
        [28, 31, 29, 30],
        [ 8, 11,  9, 10]])
```

另外一个办法就是使用 np.ix_()函数,它可以将两个一维数组转换成一个用于选取方形区域的索引器:

```
In [35]: arr[np.ix_([1,5,7,2], [0,3,1,2])]
Out[35]:
array([[ 4,  7,  5,  6],
        [20, 23, 21, 22],
        [28, 31, 29, 30],
        [ 8, 11,  9, 10]])
```

**注意**:花式索引和切片不一样,它是将数据复制到新的数组中。

### 5.1.4　数组转置和轴对换


视频讲解

转置(transpose)是重塑的一种特殊形式,它返回的是源数据的视图(不会进行任何复制操作)。数组不仅有 transpose()方法,还有一个特殊的 T 属性:

```
In [36]: arr = np.arange(15).reshape(5,3)
         Arr
Out[36]:
array([[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8],
```

```
        [ 9, 10, 11],
        [12, 13, 14]])

In [37]: arr.T
Out[37]:
array([[ 0,  3,  6,  9, 12],
       [ 1,  4,  7, 10, 13],
       [ 2,  5,  8, 11, 14]])
```

在进行矩阵计算时，经常需要用到该操作，例如利用 np.dot()计算矩阵内积：

```
In [38]: arr = np.random.randn(6,3)
np.dot(arr.T, arr)
Out[38]:
array([[  9.03630405,   0.49388948,  -1.54587135],
       [  0.49388948,   2.25164741,   1.93791071],
       [ -1.54587135,   1.93791071,  10.55460651]])
```

对于高维数组，transpose()需要得到一个由轴编号组成的元组才能对这些轴进行转置：

```
In [39]: arr = np.arange(16).reshape((2, 2, 4))
Arr
Out[39]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],

       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])
In [40]: arr.transpose((1, 0, 2))
Out[40]:
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],

       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]]])
```

### 5.1.5 利用数组进行数据处理


视频讲解

NumPy 数组可以将很多数据处理任务表述为简洁的数组表达式(否则需要编写循环)。矢量化数组运算要比 Python 方式快上一两个数

量级，尤其是对于各种数值运算。例如 np.meshgrid()函数接受两个一维数组，并产生两个二维矩阵(对应两个数组中所有的(x，y)对)。

```
In [41]: points = np.arange(-5, 5, 0.01)     #1000 个间隔相等的点
          xs, ys = np.meshgrid(points, points)
        ys
Out[41]:
array([[-5.  , -5.  , -5.  , ..., -5.  , -5.  , -5.  ],
       [-4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99],
       [-4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98],
       ...,
       [ 4.97,  4.97,  4.97, ...,  4.97,  4.97,  4.97],
       [ 4.98,  4.98,  4.98, ...,  4.98,  4.98,  4.98],
       [ 4.99,  4.99,  4.99, ...,  4.99,  4.99,  4.99]])
```

假设在一组值上计算函数 sqrt(x^2 + y^2)，这时对函数的求值运算就好办了，把这两个数组当作两个浮点数编写表达式即可：

```
In [42]: import matplotlib.pyplot as plt
          z = np.sqrt(xs ** 2 + ys ** 2)
          z
Out[42]:
array([[ 7.07106781,  7.06400028,  7.05693985, ...,  7.04988652,
          7.05693985,  7.06400028],
       [ 7.06400028,  7.05692568,  7.04985815, ...,  7.04279774,
          7.04985815,  7.05692568],
       [ 7.05693985,  7.04985815,  7.04278354, ...,  7.03571603,
          7.04278354,  7.04985815],
       ...,
       [ 7.04988652,  7.04279774,  7.03571603, ...,  7.0286414 ,
          7.03571603,  7.04279774],
       [ 7.05693985,  7.04985815,  7.04278354, ...,  7.03571603,
          7.04278354,  7.04985815],
       [ 7.06400028,  7.05692568,  7.04985815, ...,  7.04279774,
          7.04985815,  7.05692568]])
In [43]: plt.imshow(z, cmap = plt.cm.gray)
          plt.colorbar()
        plt.title('Image plot of $\sqrt{x^2 + y^2}$ for a grid of values')
Out[43]: <matplotlib.text.Text at 0x1086aa90>
```

函数值的图形化结果如图 5.1 所示。

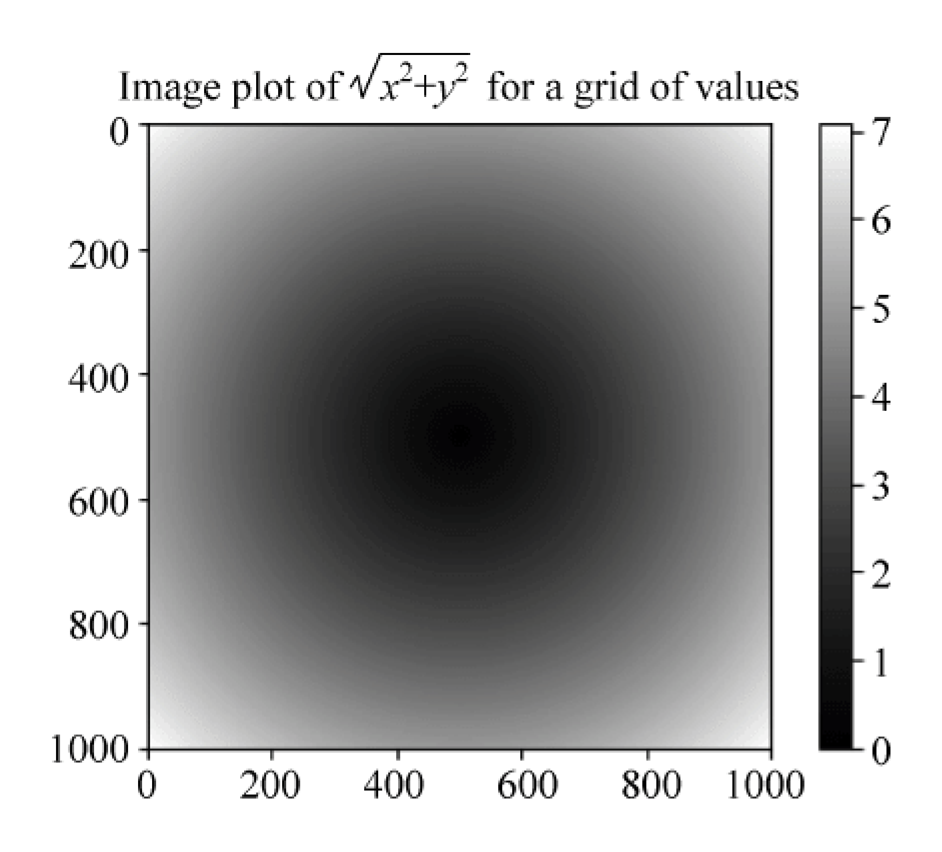



图 5.1 根据网格对函数求值的结果

### 5.1.6 数学和统计方法

用户可以通过数组上的一组数学函数对整个数组或某个轴向的数据进行统计计算。

```
In [44]: arr = np.random.randn(5, 4)        #产生正态分布数据
         arr.mean()
Out[44]: -0.24070480645161735
In [45]: np.mean(arr)
Out[45]: -0.24070480645161735
In [46]: arr.sum()
Out[46]: -4.8140961290323467
```

mean()和 sum()这类函数可以接受一个 axis 参数(用于计算该轴向上的统计值),最终结果是一个少一维的数组:

```
In [47]: arr.mean(axis = 1)
Out[47]: array([-0.26271711, -0.50185429,  0.38508322, -0.25435201, -0.56968384])
In [48]: arr.sum(0)
Out[48]: array([ 0.81837351, -2.17245972, -4.01616748,  0.55615755])
```

像 cumsum()和 cumprod()之类的方法则不聚合,而是产生一个由中间结果组成的数组:

```
In [49]: arr = np.array([[0,1,2], [3,4,5], [6,7,8]])
        arr.cumsum(0)
Out[49]:
array([[ 0,  1,  2],
```

```
        [ 3,   5,   7],
        [ 9, 12, 15]], dtype = int32)
In [50]: arr.cumprod(1)
Out[50]:
array([[   0,    0,    0],
        [   3,   12,   60],
        [   6,   42, 336]], dtype = int32)
```

## 5.2　Pandas

Pandas 的名称来自于面板数据(panel data)和 Python 数据分析(data analysis)，Pandas 是一种基于 NumPy 的数据分析包，最初由 AQR Capital Management 于 2008 年 4 月作为金融数据分析工具开发出来，并于 2009 年底开源，目前由专注于 Python 数据包开发的 PyData 开发小组继续维护。Pandas 提供了大量的高效操作大型数据集所需的函数和方法，它是使 Python 成为强大而高效的数据分析工具的重要因素之一。

### 5.2.1　Pandas 数据结构

#### 1. Series

Series 是一种类似于一维数组的对象，它由一组数据及与之相关的一组数据标签(即索引)组成。只有一组数据可产生最简单的 Series：

```
In [1]: import pandas as pd
         from pandas import Series, DataFrame
         obj = Series([4, 7,  - 5, 3])
         obj
Out[1]:
0     4
1     7
2   - 5
3     3
dtype: int64
```

Series 的字符串表现形式为索引在左边，值在右边。由于没有为数据指定索引，会自动创建一个 0～N－1(N 为数据长度)的整数型索引。用户可以通过 Series 的 values 和 index 属性获取其数组表示形式和索引对象：

```
In [2]: obj.values
Out[2]: array([ 4,  7, -5,  3], dtype=int64)
In [3]: obj.index
Out[3]: RangeIndex(start=0, stop=4, step=1)
```

通常,需要创建的 Series 带有一个可以对各个数据点进行标记的索引:

```
In [4]: obj2=Series([4,3,-5,7], index=['d','b','a','c'])
        obj2
Out[4]:
d    4
b    3
a   -5
c    7
dtype: int64
```

与普通的 NumPy 数组相比,可以通过索引的方式选取 Series 中的单个或一组值:

```
In [5]: obj2['a']
Out[5]: -5
In [6]: obj2[['c', 'a', 'd']]
Out[6]:
c    7
a   -5
d    4
dtype: int64
```

**2. DataFrame**

DataFrame 是一个表格型的数据结构,它含有一组有序的列,每列可以是不同的类型(数值型、字符串、布尔型等)。DataFrame 既有行索引,又有列索引,可以看作是由 Series 组成的字典(共用同一个索引)。跟其他类似的数据结构相比,DataFrame 中面向行和面向列的操作基本是平衡的。构建 DataFrame 的方法很多,最常见的就是直接传入一个由等长列表或 NumPy 数组组成的字典:

```
In [7]: data={'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
                'year':[2000, 2001, 2002, 2001, 2002],
                'pop':[1.5, 1.7, 3.6, 2.4, 2.9]}
    frame=DataFrame(data)
```

```
    frame
Out[7]:
   pop   state   year
0  1.5    Ohio   2000
1  1.7    Ohio   2001
2  3.6    Ohio   2002
3  2.4  Nevada   2001
4  2.9  Nevada   2002
```

如果指定了列序列，DataFrame 的列就会按照指定的顺序进行排列：

```
In [8]: DataFrame(data, columns = ['year', 'state', 'pop'])
Out[8]:
   year   state   pop
0  2000    Ohio   1.5
1  2001    Ohio   1.7
2  2002    Ohio   3.6
3  2001  Nevada   2.4
4  2002  Nevada   2.9
```

### 5.2.2　Pandas 文件操作

#### 1. Pandas 读取文件

Pandas 提供了一些用于将表格型数据读取为 DataFrame 对象的函数。表 5.2 对它们进行了总结，其中 read_csv()和 read_table()可能会是今后用得最多的。

表 5.2　Pandas 读取文件的函数

| 函　　数 | 说　　明 |
| --- | --- |
| read_csv() | 从文件、URL、文件型对象中加载带分隔符的数据。默认分隔符为逗号 |
| read_table() | 从文件、URL、文件型对象中加载带分隔符的数据。默认分隔符为制表符("\t") |
| read_fwf() | 读取定宽列格式数据(也就是说没有分隔符) |
| read_clipboard() | 读取剪贴板中的数据，可以看作是 read_table()的剪贴板。它在将网页转换为表格时非常有用 |

#### 2. Pandas 导出文件

Pandas 导出文件的函数如表 5.3 所示。

表 5.3　Pandas 导出文件的函数

| 函　　数 | 说　　明 |
| --- | --- |
| to_csv(file_path,sep=' ',index=True,header=True) | file_path 表示文件路径<br>sep 表示分隔符<br>index 代表是否导出行序号<br>header 代表是否导出列序号 |
| to_excel(file_path,sep=' ',index=True,header=True) | file_path 表示文件路径<br>sep 表示分隔符<br>index 代表是否导出行序号<br>header 代表是否导出列序号 |

### 5.2.3　数据处理

在数据分析中，数据清洗是数据价值链中最关键的步骤。数据清洗就是处理缺失数据以及清除无意义的信息。对于垃圾数据，即使是通过最好的分析，也将产生错误的结果，并误导业务本身。

对缺失值的处理有数据补齐、删除对应行、不处理等几种方法。例如：

```
In [9]:
import pandas as pd
import numpy as np
from pandas import DataFrame
data = {'Tom':[170, 26, 30],'Mike':[175, 25, 28],'Jane':[170, 26,np.nan],'Tim':[175, 25, 28]}
data1 = DataFrame(data).T
data1.drop_duplicates()
data1
```

该段代码的输出结果如下：

```
Out [9]:
          0      1      2
Jane   170.0   26.0    NaN
Mike   175.0   25.0   28.0
Tim    175.0   25.0   28.0
Tom    170.0   26.0   30.0
```

方法一：删除有缺失值的行。

```
In [10]:
data2 = data1.dropna()
data2
```

删除后的结果如下：

```
Out [10]:
          0     1     2
Mike  175.0  25.0  28.0
Tim   175.0  25.0  28.0
Tom   170.0  26.0  30.0
```

通过使用 dropna()方法后，可以看到第 1 行存在缺失值，故被删掉了。

方法二：对缺失值进行填充有很多方法，比较常用的有均值填充、中位数填充、众数填充等。以下采用均值填充：

```
In [11]:
data3 = data1.fillna(data1.mean())
data3
```

结果为：

```
Out [11]:
          0     1          2
Jane  170.0  26.0  28.666667
Mike  175.0  25.0  28.000000
Tim   175.0  25.0  28.000000
Tom   170.0  26.0  30.000000
```

### 5.2.4　层次化索引

层次化索引是 Pandas 的一个重要功能，它能使一个轴上有多个(两个以上)索引级别，即它能以低维度形式处理高维度数据。

```
In [12]:
import pandas as pd
import numpy as np
from pandas import Series,DataFrame
data = Series(np.random.randn(10),
              index = [['a','a','a','b','b','b','c','c','d','d'],
                       [1,2,3,1,2,3,1,2,2,3]])
data
Out[12]:
a  1   -0.088594
   2    0.316611
   3    1.383978
```

```
b  1     0.215510
   2   - 0.111913
   3   - 0.580355
c  1   - 0.048050
   2   - 0.054285
d  2   - 0.136860
   3   - 1.578472
dtype: float64
```

这就是带有多重索引的 Series 格式化输出。下面看一下它的索引：

```
In [13]:
data.index
Out[13]:
MultiIndex(levels = [['a', 'b', 'c', 'd'], [1, 2, 3]],
           labels = [[0, 0, 0, 1, 1, 1, 2, 2, 3, 3], [0, 1, 2, 0, 1, 2, 0, 1, 1, 2]])
```

对于一个层次化索引的对象，选取一个数据集很简单：

```
In [14]:
data['b']
Out[14]:
1     0.215510
2   - 0.111913
3   - 0.580355
In [15]:
data['b':'c']
Out[15]:
b  1     0.215510
   2   - 0.111913
   3   - 0.580355
c  1   - 0.048050
   2   - 0.054285
dtype: float64
In [16]:
data[['b','d']]
Out[16]:
b  1     0.215510
   2   - 0.111913
   3   - 0.580355
d  2   - 0.136860
   3   - 1.578472
dtype: float6
```

甚至还可以在“内层”中进行选取：

```
In [17]:
data[:,2]
Out[17]:
a    0.316611
b   -0.111913
c   -0.054285
d   -0.136860
dtype: float64
```

层次化索引在数据重塑和基于分组的操作中扮演着重要的角色。例如，一个数据可以通过它的 unstack()方法被重新安排到一个 DataFrame 中：

```
In [18]:
data.unstack()
Out[18]:
          1          2          3
a -0.088594   0.316611   1.383978
b  0.215510  -0.111913  -0.580355
c -0.048050  -0.054285        NaN
d       NaN  -0.136860  -1.578472
```

对于一个 DataFrame 对象，每条轴都可以有分层索引：

```
In [19]:
df = DataFrame(np.arange(12).reshape((4,3)),
               index = [['a','a','b','b'],[1,2,1,2]],
                  columns = [['Ohio','Ohio','Colorado'],
                             ['green','red','green']])
Df
Out[19]:
     Ohio      Colorado
    green red     green
a 1     0   1         2
  2     3   4         5
b 1     6   7         8
  2     9  10        11
```

各层都可以有名字(字符串或者其他 Python 对象)。如果指定名称，它就会显示在控制台输出中：

```
In [20]:
df.index.names = ['key1','key2']
df.columns.names = ['state','color']
df
```

```
Out[20]:
state      Ohio     Colorado
color      green red    green
key1 key2
a    1        0   1          2
     2        3   4          5
b    1        6   7          8
     2        9  10         11
```

由于有了分部的索引，所以可以很轻松地选取列分组：

```
In [21]:
df['Ohio']
Out[21]:
color       green  red
key1 key2
a    1         0     1
     2         3     4
b    1         6     7
     2         9    10
```

### 5.2.5 分级顺序

#### 1. 重新分级排序

有时需要重新调整某条轴上各级别的顺序，或根据指定级别的值对数据进行重新排序。Swaplevel()接受两个级别编号或名称，并返回一个互换级别的新对象(但数据不会发生变化)。

```
In [22]:
df.swaplevel('key1', 'key2')
Out[22]:
state      Ohio     Colorado
color      green red    green
key2 key1
1    a        0   1          2
2    a        3   4          5
1    b        6   7          8
2    b        9  10         11
```

在交换级别时经常会用到 sortlevel()，sortlevel()根据单个级别中的值对数据进

行排序,这样最终结果就是有序的了:

```
In [23]:
df.sortlevel(1)
Out[23]:
state       Ohio      Colorado
color     green red     green
key1 key2
a    1         0   1          2
b    1         6   7          8
a    2         3   4          5
b    2         9  10         11
```

### 2. 根据级别汇总统计

许多对 DataFrame 和 Series 的描述和汇总统计都有一个 level 选项,它用于指定在某条轴上求和的级别。例如:

```
In [24]:
df.sum(level = 'key2')
Out[24]:
state   Ohio      Colorado
color green red     green
key2
1          6   8        10
2         12  14         16
```

## 5.2.6　使用 DataFrame 的列

有时希望将 DataFrame 的一个或多个列索引当成行用,或者将 DataFrame 的行索引变成列。

```
In [25]:
Df = DataFrame({'a':range(7),'b':range(7,0, - 1),
                'c':['one','one','one','two','two','two','two'],
                'd':[0,1,2,0,1,2,3]})
Df
Out[25]:
   a  b    c  d
0  0  7  one  0
1  1  6  one  1
2  2  5  one  2
```

```
3   3   4   two   0
4   4   3   two   1
5   5   2   two   2
6   6   1   two   3
```

DataFrame 的 set_index()方法会将一个或多个列转化成行索引,并创建一个新的 DataFrame:

```
In [26]:
df1 = df.set_index(['c','d'])
df1
Out[26]:
         a   b
c    d
one 0    0   7
     1   1   6
     2   2   5
two 0    3   4
     1   4   3
     2   5   2
     3   6   1
```

DataFrame 的 reset_index()方法会将层次化索引的级别转移到列里面去:

```
In [27]:
df1.reset_index()
Out[27]:
      c    d   a   b
0   one    0   0   7
1   one    1   1   6
2   one    2   2   5
3   two    0   3   4
4   two    1   4   3
5   two    2   5   2
6   two    3   6   1
```

## 5.3 Matplotlib

Matplotlib 可以通过绘图帮助用户找出异常值,进行必要的数据转换,得出有关模型的 idea 等,是 Python 数据分析重要的可视化工具。

### 5.3.1　figure 和 subplot

Matplotlib 的图像都位于 figure 中，可以用 plt.figure()创建一个新的 figure：

```
In [28]:
import matplotlib.pyplot as plt
fig = plt.figure()            # 创建一个新的 figure，会弹出一个空窗口
```

plt.figure()的一些选项，特别是 figuresize，可以确保图片保存到磁盘上时具有一定的大小和纵横比。plt.gcf()可得到当前 figure 的引用，必须用 add_subplot()创建一个或多个 subplot 才可以绘图：

```
In [29]:
ax1 = fig.add_subplot(2,2,1)
```

以上代码的意思是该图像是 2×2 的（即有 4 个 subplot），且当前选中的是 4 个 subplot 中的第一个（编号从 1 开始）。如果要把后面的也创建并显示出来，可以用如下代码：

```
In [30]:
ax2 = fig.add_subplot(2,2,2)
ax3 = fig.add_subplot(2,2,3)
ax4 = fig.add_subplot(2,2,4)
```

这几行代码运行的结果如图 5.2 所示。

```
Out[30]:
```

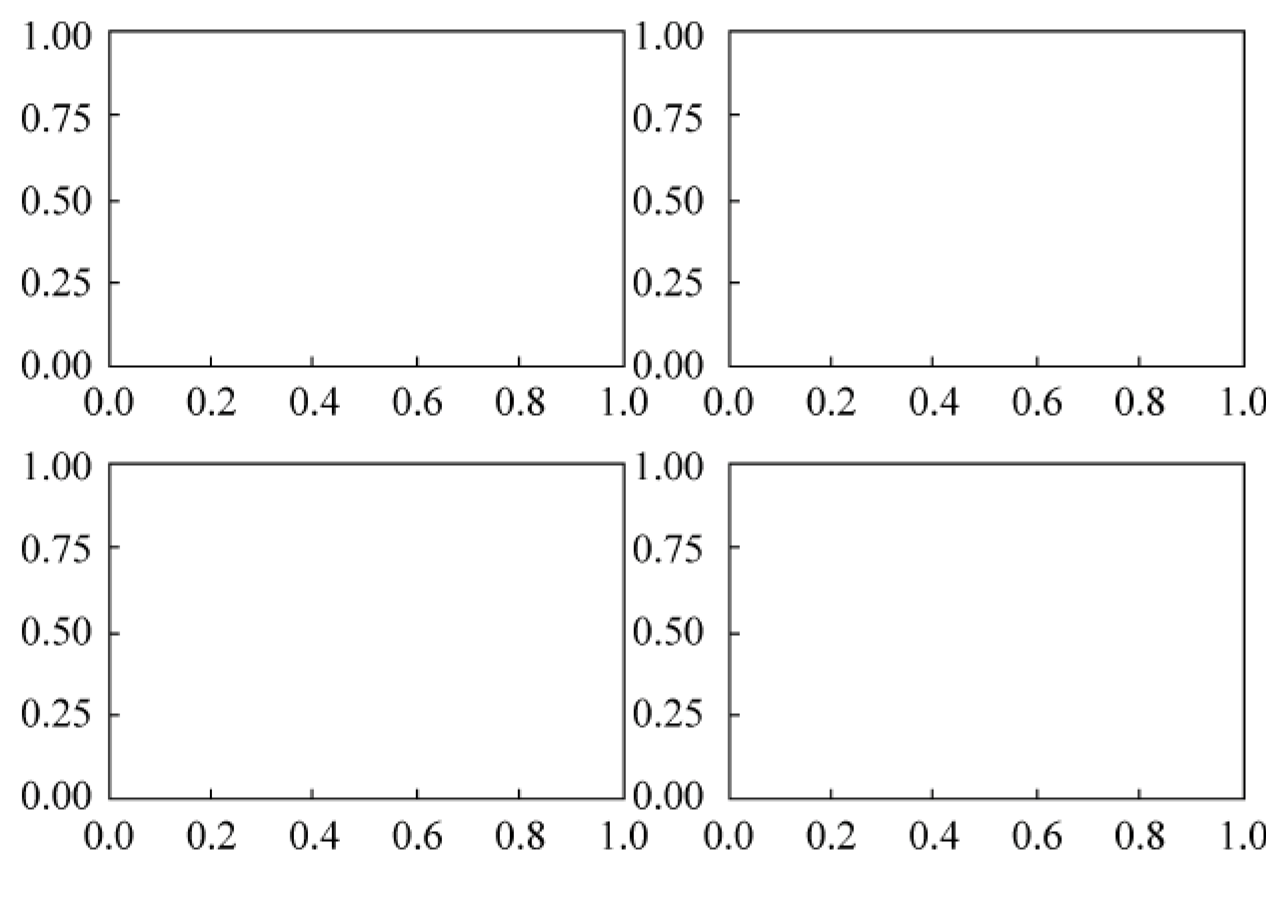



图 5.2　有 4 个 subplot 的 figure

这时如果执行一条绘图命令 plt. plot([ ]),Matplotlib 就会在最后一个用过的 subplot(没有则创建一个)上进行绘制。因此,执行下列代码可以得到图 5.3 所示的结果:

```
In [31]:
from numpy.random import randn
plt.plot(randn(50).cumsum(), 'k-- ')
Out[31]:
```

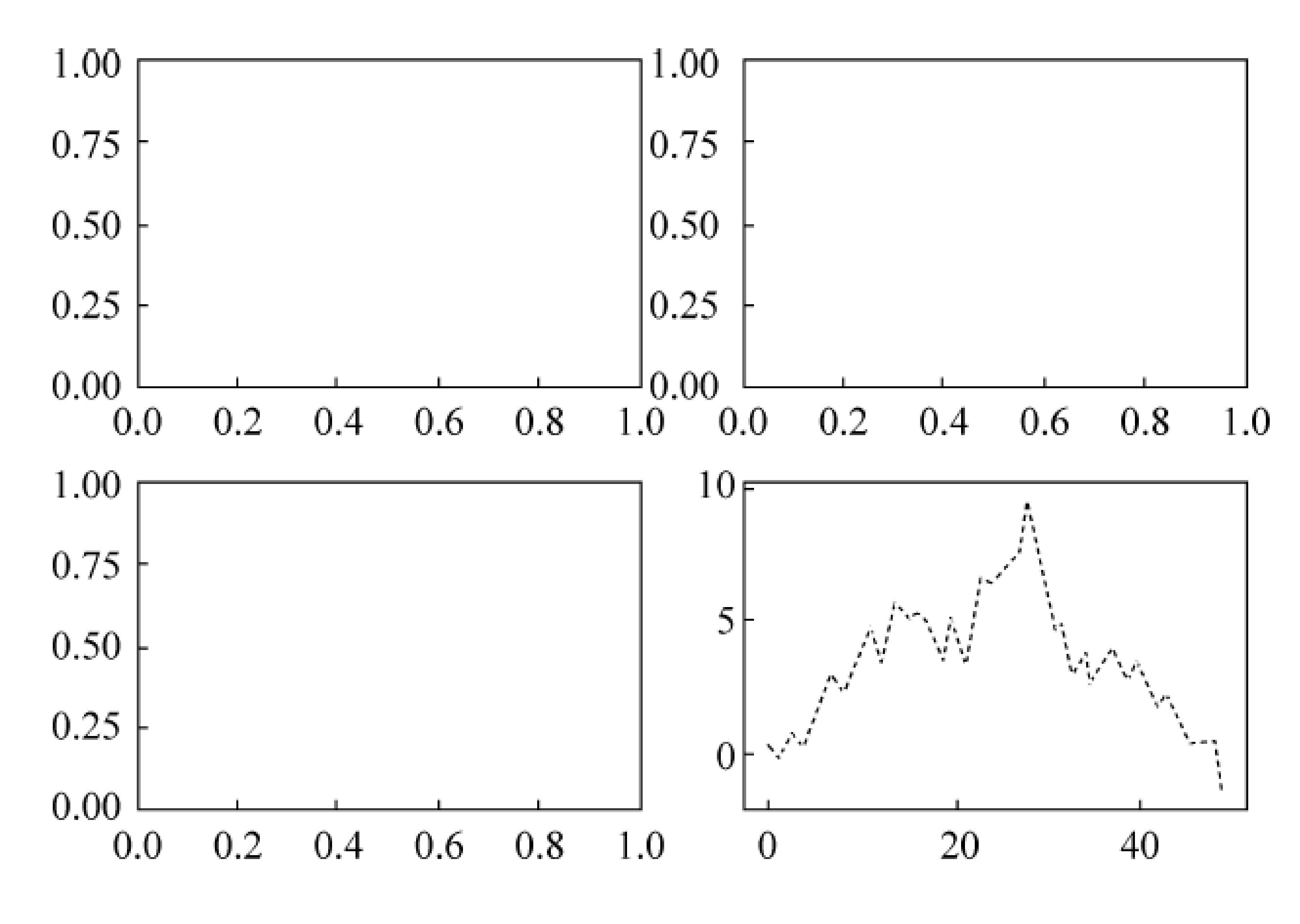



图 5.3 进行绘制操作后的图

'k--'是一个线型选项,用于告诉 Matplotlib 绘制黑色虚线图。前面那些由 fig. add_subplot()返回的是 AxesSubplot 对象,直接调用其实例方法就可以在其他空着的格子里面绘图:

```
In [32]:
import numpy as np
ax1.hist(randn(100),bins = 20,color = 'k', alpha = 0.3)
ax2.scatter(np.arange(30),np.arange(30) + 3 * randn(30))
```

结果如图 5.4 所示。

```
Out[32]:
```

可以在 Matplotlib 中找到各种图标类型。根据特定布局创建 figure 和 subplot 是一件非常常见的任务,于是便出现了一个更为方便的方法——plt. subplots()。它可以创建一个新的 figure,并返回一个含有已创建的 subplot 对象的 NumPy 数组:

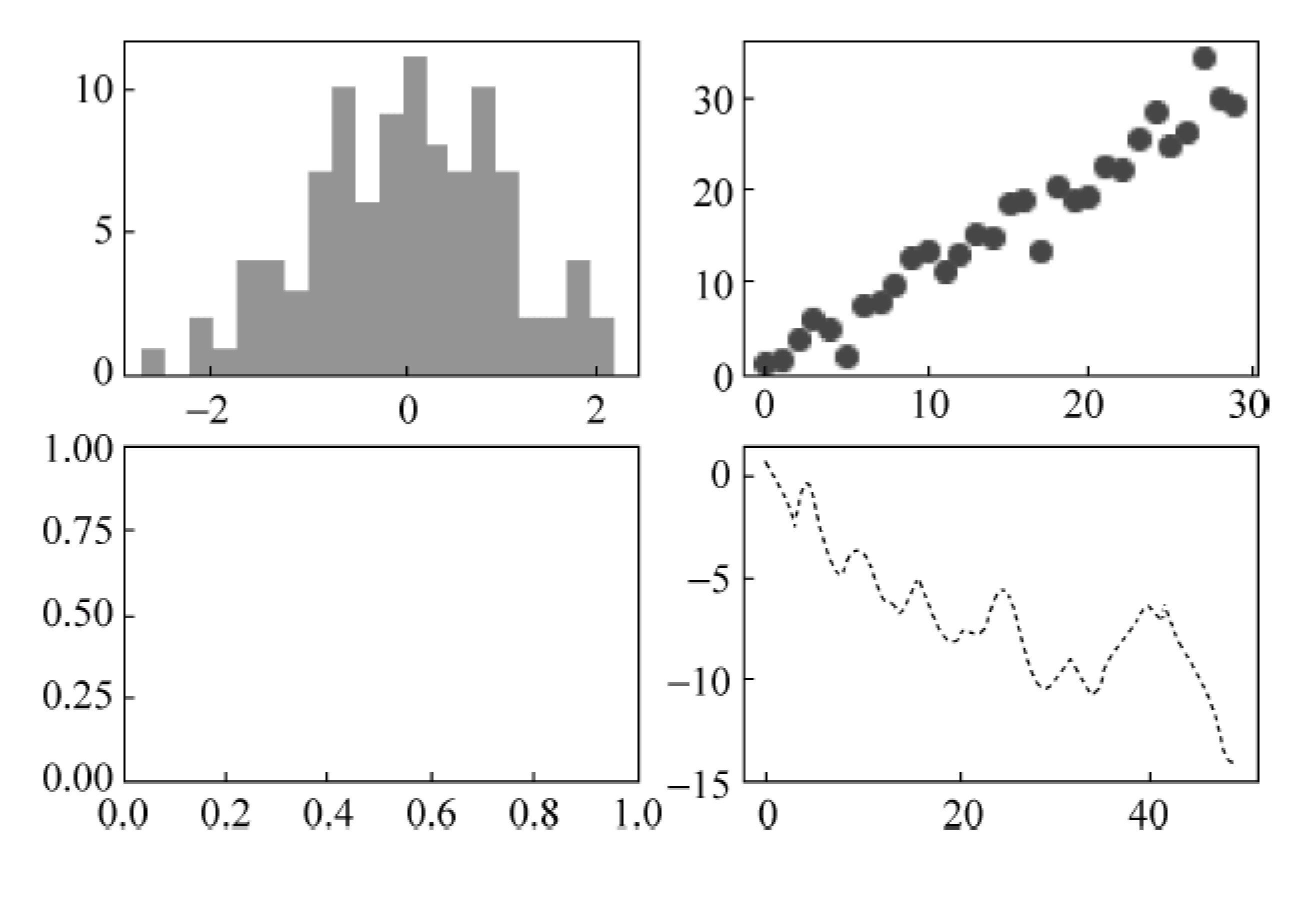



图 5.4 连续绘制后的图

```
In [33]:
fig, axes = plt.subplots(2,3)
axes
```

输出结果为：

```
Out[33]:
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x0000000012486278>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x0000000013EFF780>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x00000000161A67B8>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x00000000161FF588>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x0000000016265AC8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x00000000162BE400>]],
       dtype=object)
```

这是非常实用的，因为可以轻松地对 axes 数组进行索引，就好像一个二维数组一样，例如 axes[0,1]。用户还可以通过 sharex 和 sharey 指定 subplot 应该具有相同的 X 轴或 Y 轴。在比较相同范围内的数据时，这也是非常实用的，否则 Matplotlib 会自动缩放各图表的界限。关于 subplots 的更多信息如表 5.4 所示。

**表 5.4 pyplot.subplots()的参数**

| 参　数 | 说　明 |
|---|---|
| nrows | subplot 的行数 |
| ncols | subplot 的列数 |
| sharex | 所有 subplot 应该使用相同的 X 轴刻度(调节 xlim 会影响所有的 subplot) |
| sharey | 所有 subplot 应该使用相同的 Y 轴刻度(调节 ylim 会影响所有的 subplot) |

续表

| 参　　数 | 说　　明 |
|---|---|
| subplot_kw | 用于创建各 subplot 的关键字字典 |
| ** fig_kw | 创建 figure 时的其他关键字 |

### 5.3.2 调整 subplot 周围的间距

在默认情况下，Matplotlib 会在 subplot 外围留下一定的边距，并在 subplot 之间留下一定的间距。间距跟图像的高度和宽度有关，因此，如果调整了图像大小，间距也会自动调整。利用 figure 的 subplots_adjust()方法可以轻而易举地修改间距，代码如下：

```
In [34]:
fig, axes = plt.subplots(2,2,sharex = True, sharey = True)
for i in range(2):
    for j in range(2):
        axes[i,j].hist(randn(500),bins = 50, color = 'k', alpha = 0.5)
plt.subplots_adjust(wspace = 0, hspace = 0)
```

wspace 和 hspace 用于控制宽度和高度的百分比，可以用作 subplots 之间的间距，在这个例子中将间距收缩到 0，如图 5.5 所示。

```
Out[34]:
```

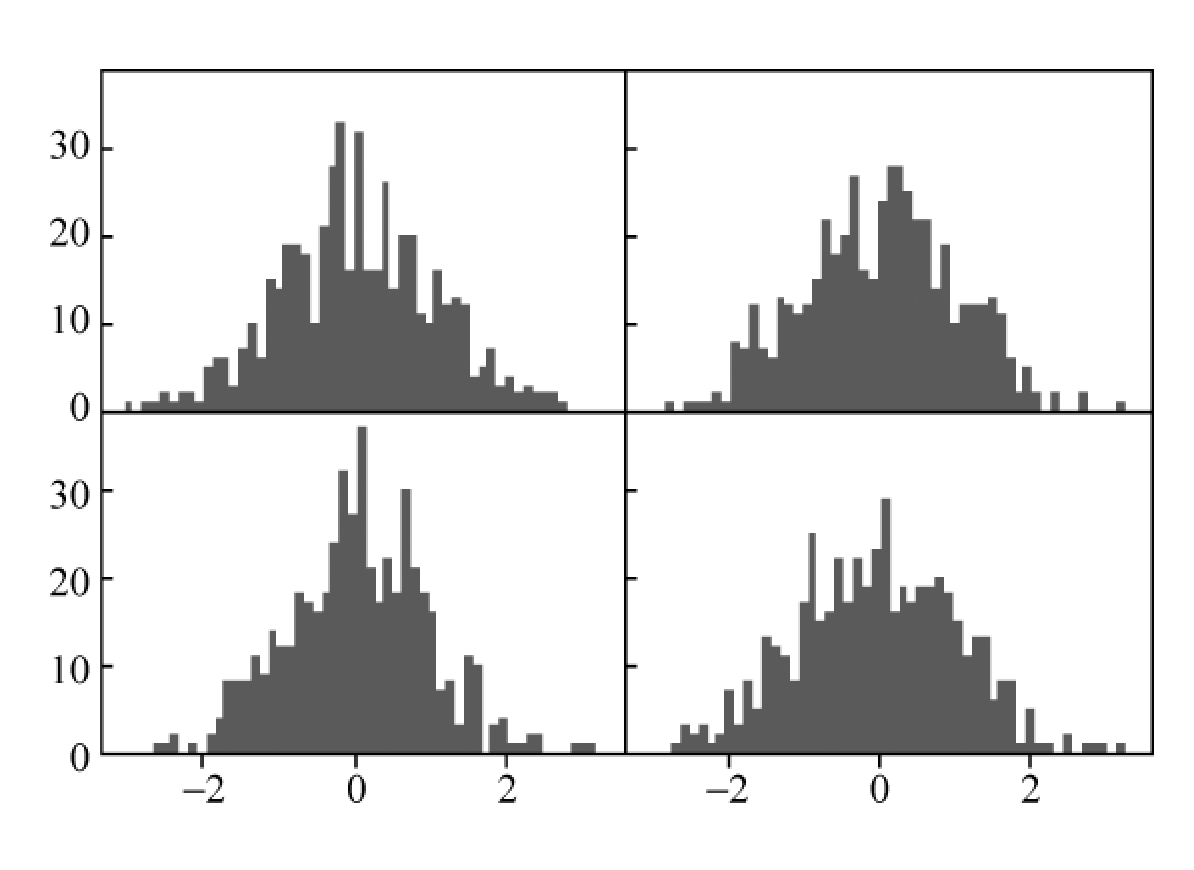



图 5.5　各 subplots 之间没有间距

由图 5.5 不难看出其中的轴标签重叠了。Matplotlib 不会检查轴标签是否重叠，所以对于这种情况，用户只能自己设定刻度位置和刻度标签。

### 5.3.3　颜色、标记和线型

Matplotlib 的 plot()函数可以接受一组(x,y)坐标以及表示颜色和线型的字符串缩写。常用的颜色都有一个缩写词,如果要使用其他颜色,可以使用指定其 RGB 值的方式。例如要根据 x 和 y 绘制红色虚线,可以执行如下代码:

```
In [35]:
plt.plot(x,y,'r--')
```

这种在一个字符串中指定颜色和线型的方式非常方便,也可以通过下面这种更为明确的方式得到同样的效果:

```
In [36]:
plt.plot(x,y,linestyle='--', color='r')
```

### 5.3.4　刻度标签和图例

对于大多数的图标装饰项而言,其实现方式主要有两种,即使用过程型的 pyplot 接口和更为面向对象的原生 Matplotlib API。设计 pyplot 接口的目的就是实现交互式作用,它含有诸如 xlim()、xticks()和 xticklabels()之类的方法,分别控制图表的范围、刻度位置和刻度标签等。其使用方式有以下两种:

(1) 调用时不带参数,则返回当前的参数值。例如,plt. xlim()返回当前 X 轴的绘图范围。

(2) 调用时带参数,则设置参数。例如,plt. xlim([0,100])会将 X 轴的范围设置为0～100。

这些方法都是对当前或最近创建的 AxesSubplot 起作用,它们各自对应 subplot 对象上的两个方法。以 xlim()为例,就是 ax. get_xlim()和 ax. set_xlim()。为了说明轴的自定义,创建一个简单的图像并绘制一段随机漫步图,如图 5.6 所示:

```
In [37]:
fig=plt.figure()
ax=fig.add_subplot(1,1,1)
ax.plot(randn(1000).cumsum())
```

```
Out[37]:
```

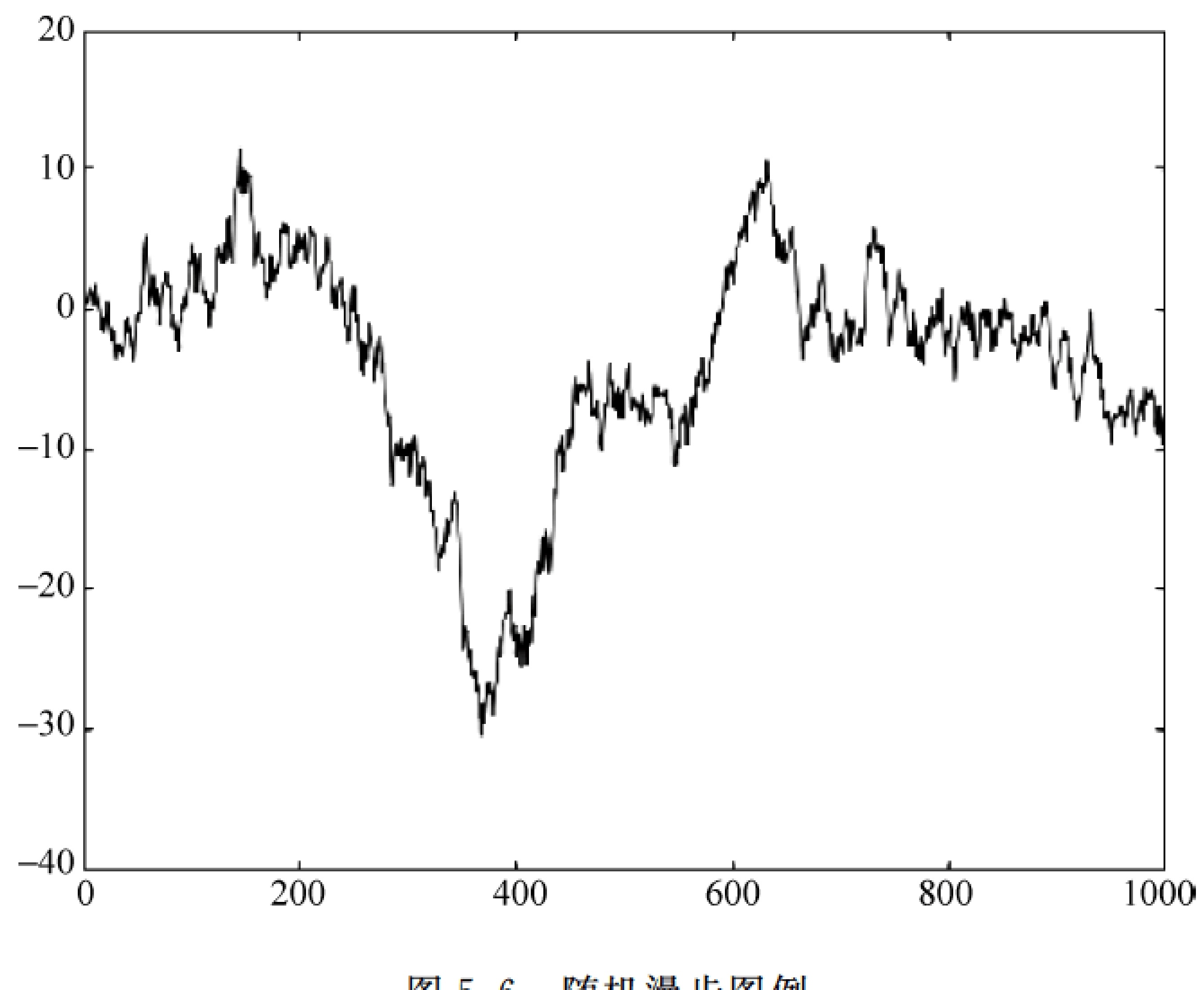



图 5.6　随机漫步图例

如果要修改 X 轴刻度,最简单的办法是使用 set_xticks()和 set_xticklabels()。前者告诉 Matplotlib 要将刻度放在数据范围中的哪些位置,在默认情况下这些位置也就是刻度标签。

### 5.3.5　添加图例

图例(legend)是另外一种用于表示图标元素的重要工具。添加图例的最简单的方式,就是在添加 subplot 的时候传入 label 参数:

```
In [38]:
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.plot(randn(1000).cumsum(),'k',label = 'one')
```

当需要对图中的线进行注解时,可用下面这样的代码添加图例:

```
In [39]:
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.plot(randn(1000).cumsum(),'k',label = 'one')
ax.plot(randn(1000).cumsum(),'k-- ',label = 'one')
```

```
ax.plot(randn(1000).cumsum(),'k.',label = 'one')
ax.legend(loc = 'best')
```

这几行代码得到的效果如图 5.7 所示。用户可以通过 loc 参数来指定图例所在的位置,'best'表示它会自动找一个最佳位置。

```
Out[39]:
```

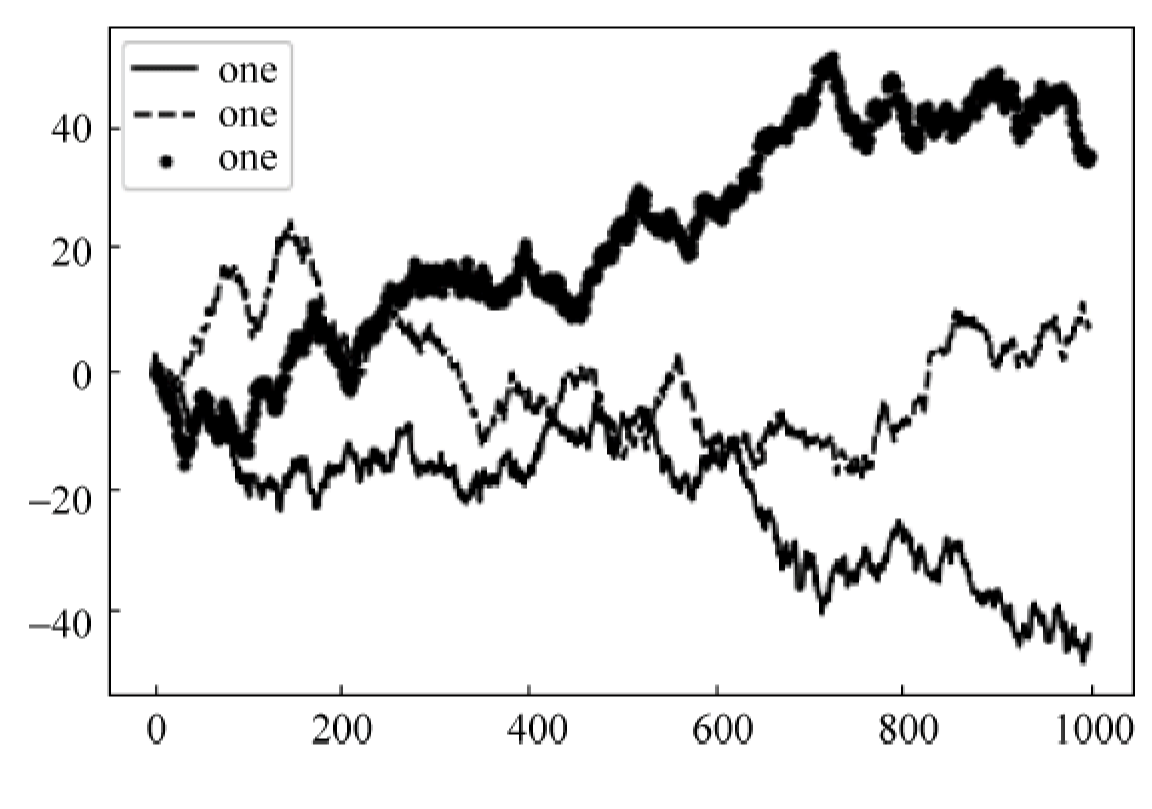



图 5.7　在最佳位置添加图例

### 5.3.6　将图表保存到文件

利用 plt.savefig()方法可以将当前图表保存到文件。该方法相当于 figure 对象的 savefig()实例方法。例如要将图表保存为 SVG 格式文件,需要用如下代码:

```
In [40]:
plt.savefig('figpath.svg')
```

文件类型是通过文件扩展名推断出来的。因此,如果用户使用的是.jpg,就会得到一个 JPG 格式的文件。在发布图片时最常用到的两个选项是 dpi(控制"每英寸点数")和 bbox_inches(剪除当前图表周围的空白部分)。如果用户想得到一个指定分辨率的文件,可以用下面的语句:

```
In [41]:
plt.savefig('figpath.svg', dpi = xxx, bbox_inches = 'tight')
```

dpi 表示想要得到的分辨率，bbox_inches= 'tight'表示得到的图片带有最小的白边。figure. savefig()方法的参数说明如表 5.5 所示。

**表 5.5 figure. savefig()方法的参数说明**

| 参　数 | 说　明 |
|---|---|
| fname | 含有文件路径的字符串，或者 Python 的文件型对象，图像格式由文件扩展名推断而出 |
| dpi | 图像的分辨率(每英寸点数)，默认等于 100 |
| facecolor | 图像的背景颜色，默认为白色 |
| egdecolor | 图像四周的颜色，默认为白色 |
| format | 设置文件格式，例如 png、pdf、svg、jpg…… |
| bbox_inches | 图像需要保存的部分。如果设置为 'tight'，则会尝试剪掉图像周围的空白部分 |

## 5.4 SciPy

SciPy 建立在 NumPy 基础之上，集成了众多的数学、科学以及工程计算中常用库函数的 Python 模块，例如线性代数、常微分方程数值求解、信号处理、图像处理、稀疏矩阵等。通过给用户提供一些高层的命令和类，SciPy 在 Python 交互式会话中大大增加了操作和可视化数据的能力。通过 SciPy，Python 的交互式会话变成了一个数据处理和 system-prototyping 的环境，足以和 MATLAB、IDL、Octave、R-Lab 以及 SciLab 抗衡。

SciPy 的子模块涵盖了不同科学计算领域的内容，表 5.6 对它们进行了总结。

**表 5.6 SciPy 子模块的描述**

| 子　模　块 | 描　述 |
|---|---|
| constans | 物理和数学常数 |
| cluster | 聚类算法 |
| fftpack | 快速傅里叶变换程序 |
| integrate | 集成和常微分方程求解器 |
| interpolate | 拟合和平滑曲线 |
| io | 输入和输出 |
| linalg | 线性代数 |
| maxentropy | 最大熵法 |
| ndimage | N 维图像处理 |

续表

| 子　模　块 | 描　　述 |
|---|---|
| odr | 正交距离回归 |
| optimize | 最优路径选择 |
| signal | 信号处理 |
| sparse | 稀疏矩阵以及相关程序 |
| spatial | 空间数据结构和算法 |
| special | 特殊函数 |
| states | 统计上的函数和分布 |
| weave | C/C++整合 |

例如，用 optimize 实现最优化：

```
In [42]:
from scipy import *
import matplotlib.pyplot as plt
import numpy as np
from scipy import optimize
#最优化问题(寻找函数的最大值或者最小值)是数学中的一大领域,复杂函数的最优化问题
#或者多变量的最优化问题,可能会非常复杂
x = linspace( - 5, 3, 100)
def f(x):
    return 4 * x * * 3 + (x - 2) * * 2 + x * * 4
#局部最小值
x_min_local = optimize.fmin_bfgs(f, 2)
print(x_min_local)
#全局最小值
x_max_global = optimize.fminbound(f, - 10, 10)
print(x_max_global)
```

## 5.5　Scikit-learn

Scikit-learn(Sklearn)是 Python 基于 NumPy、SciPy、Matplotlib 实现机器学习的算法库，Scikit-learn 库始于 2007 年的 Google Summer of Code 项目，最初由 David Cournapeau 开发。它是一个简洁、高效的算法库，可以实现数据预处理、分类、回归、降维、模型选择等常用的机器学习算法，以用于数据挖掘和数据分析，具体内容见第 10 章。图 5.8 所示为 Scikit-learn 算法框架图。

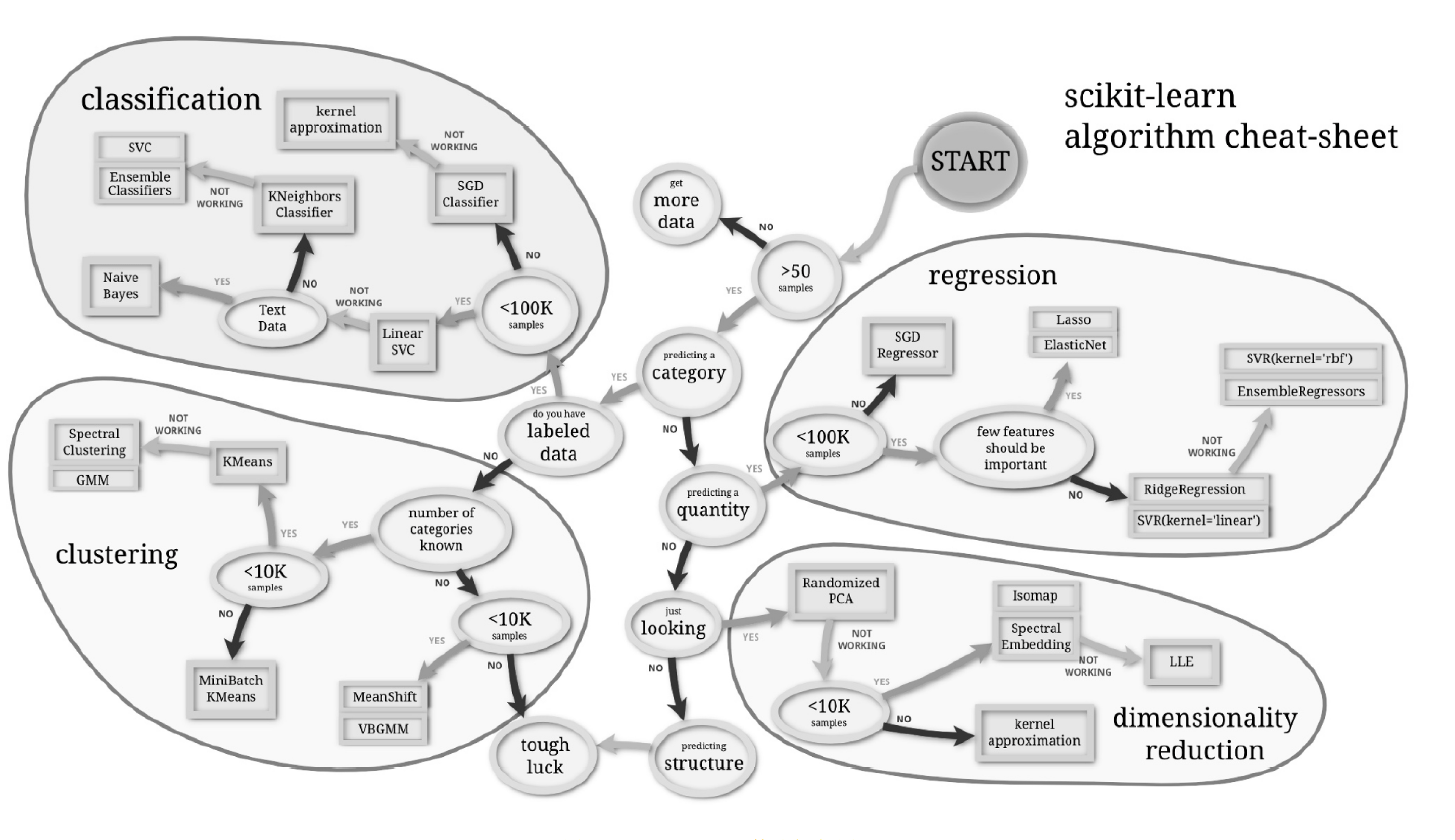



图 5.8　Scikit-learn 算法框架图

## 本章小结

本章介绍 Python 数据分析的常用库：NumPy 数值计算库是数据分析的基础，它将数据转化为数组进行计算；Pandas 是 Python 数据分析的标准库，里面包含了很多数据分析的工具；Matplotlib 是将数据可视化的库，可以让用户对数据有一个更加直观、清晰的认识；SciPy 是一个基于 NumPy 的集成数学计算库；Scikit-learn 是一个集成了很多机器学习算法的库。

## 习题

1. 创建一个长度为 10 的一维的全为 0 的 ndarray 对象，然后让第 3 个元素等于 5。

2. 利用 Matplotlib 画出一个 1000 步的随机漫步(Random Walk)的图例，通过 set_xticks()和 set_xticklabels()将其放在最佳位置。

3. 根据如下原始数据集 raw_data 生成一个 DataFrame，并将其赋值给变量 army。

raw_data={'regiment'：['Nighthawks'，'Nighthawks'，'Nighthawks'，'Nighthawks'，'Dragoons'，'Dragoons'，'Dragoons'，'Dragoons'，'Scouts'，'Scouts'，'Scouts'，'Scouts']，'company'：['1st'，'1st'，'2nd'，'2nd'，'1st'，'1st'，'2nd'，'2nd'，'1st'，'1st'，'2nd'，'2nd']，'deaths'：[523，52，25，616，43，234，523，62，62，73，37，35]，'battles'：[5，42，2，2，4，7，8，3，4，7，8，9]，'size'：[1045，957，1099，1400，1592，1006，987，849，973，1005，1099，1523]，'veterans'：[1，5，62，26，73，37，949，48，48，435，63，345]，'readiness'：[1，2，3，3，2，1，2，3，2，1，2，3]，'armored'：[1，0，1，1，0，1，0，1，0，0，1，1]，'deserters'：[4，24，31，2，3，4，24，31，2，3，2，3]，'origin'：['Arizona'，'California'，'Texas'，'Florida'，'Maine'，'Iowa'，'Alaska'，'Washington'，'Oregon'，'Wyoming'，'Louisana'，'Georgia']}

# 第 6 章



# 网络数据的获取

**本章学习目标：**

- 了解网页的基本组织形式
- 了解 HTML 和 XML 的基础知识
- 熟练掌握 urllib 和 BeautifulSoup4 模块

随着互联网在世界范围内的迅速普及，其产生了越来越多的数据。互联网自出现以来，与人类社会各方面的发展联系得越来越紧密，其产生的数据中蕴藏着大量对社会生产实践过程有用的信息。但是，数据量的急剧增加也带来了一个严重的问题——信息过载。通俗地理解，就是单凭人力难以完成人们所需数据的检索与获取。换而言之，即无法凭借人有限的计算力从海量的数据中获取有用的信息。幸运的是随着计算机技术的飞快发展，人们开始可以借助计算机代替或者协助完成这项繁重的任务。本章是为获取网络数据做前期准备，在内容上限于互联网网页数据的获取。本章首先介绍两种流行的互联网网页形式——HTML 和 XML；然后介绍如何使用 urllib 获取网页数据；最后介绍如何使用 BeautifulSoup4 解析网页文档。

# 6.1　网页数据的组织形式

将网页作为一个整体进行获取并不困难，困难之处在于如何从网页中提取出用户所需要的数据。想要实现这个目的，了解网页是如何组织的就显得非常必要。在浏览器中，用户可以看见网页的最终呈现形式，继而很清楚地知道自己需要哪些数据。那么如何通过计算机获取这些数据呢？通常而言，计算机程序获取的是以文本形式存在的网页源代码，必须由用户告诉它需要提取网页源代码中的哪部分数据。本节简要介绍两种典型的网页组织形式，即 HTML 和 XML，从而为网络数据的抓取做好基础工作。

## 6.1.1　HTML

HTML 的英文全称是 Hyper Text Markup Language，中文译为超文本标记语言。超文本标记语言是标准通用标记语言下的一个应用，也是一种规范、一种标准，它通过标记符号来标记要显示的网页中的各个部分。网页文件本身是一种文本文件，通过在文本文件中添加标记符可以告诉浏览器如何显示其中的内容（例如文字如何处理、画面如何安排、图片如何显示等）。浏览器按顺序阅读网页文件，然后根据标记符解释和显示其标记的内容，但对书写出错的标记不指出错误，且不停止其解释执行过程。程序编写者只能通过显示效果来分析出错原因和出错位置。需要注意的是，不同的浏览器对同一个标记符可能会有不完全相同的解释，因此可能会呈现出不同的显示效果。

超文本标记语言文档的制作并不是很复杂，但其功能确实异常强大，支持不同数据格式的文件放入 HTML 文档中，这也是万维网（WWW）盛行的原因之一。HTML 主要有以下几个特点。

（1）可扩展性：超文本标记语言的广泛应用促使越来越多的组织或者个人为其带来了加强功能、增加标识符等要求，超文本标记语言采取子类元素的方式，为系统扩展带来保证。

（2）平台无关性：虽然安装有 Windows 操作系统的计算机盛行于世，但使用 MacOS、Linux、UNIX 等其他操作系统计算机的也大有人在。超文本标记语言是一

项互联网通用标准，可以广泛使用在各种平台上，并不依赖于某个或某些特定的操作系统。这也是万维网盛行的另一个原因。

(3) 通用性：超文本标记语言是网络的通用语言，是一种简单、通用的全置标记语言。其允许网页制作者基于 HTML 建立文本与图片、音频、视频相结合的复杂页面，这些页面可以被网上的任何人浏览到，而无论他们使用的是什么类型的操作系统或浏览器。

HTML 文档包含了 HTML 标签(TAG)和文本，通过它们来描述网页。Web 浏览器的作用是将 HTML 源文件转化成网页形式，并显示出它们。浏览器本身并不会显示出 HTML 标签，而是使用它们来解释页面的内容。

### 6.1.2 HTML 元素

HTML 标签通常用两个角括号括起来，即用< >来进行标记，例如段落标记< p >、图片标记< img >等。标签都是闭合的(两种格式：双标签(成对)与单标签(不成对))。双标签形如“<标签名>标签内容</标签名>”，例如< table >…</table >。这被称为一个标签对，并分为开始标签和结束标签。第一个标签是开始标签(也称为开放标签)，第二个标签是结束标签(也称为闭合标签)。单标签形如“<标签名/>”，例如< br/>、< hr/>等。在 HTML 中，大多数 HTML 元素通常都是成对出现的。这里有两点需要注意：第一，由于 HTML 是一种弱势语言，如果单标签不写“/”一般也不会报错，但如果放在某些规定比较严格的浏览器上运行，可能会出现问题，所以建议用户按照规范的格式写代码；第二，标签是大小写无关的，例如< body >和< BODY >表示的意思是一样的，推荐用户使用小写，这样符合 XHTML 标准。开始标签和结束标签以及它们之间包含的内容构成了一个元素。某些 HTML 元素没有内容，这被称为空元素。空元素在开始标签中进行关闭(以开始标签的结束而结束)。

在大多数元素之中可以嵌套其他元素。例如，html 元素< html >…</html >之间可以嵌套主体元素< body >…</body >，而主体元素< body>…</body >之间又可以嵌套段落元素< p >…</p >。对于 HTML 文档，嵌套是它最基本的组织架构之一。对于 HTML 元素，有以下几个需要注意的问题。

(1) 结束标签：对于目前的 HTML 版本来说，即使用户忘记使用结束标签，大多数浏览器也会正确地显示相应的 HTML 内容，但是并不建议这样做。在未来的

HTML 版本中将逐步严格要求使用结束标签。

(2) 空元素：没有内容的 HTML 元素称为空元素。它并不需要通过类似于 < tag >…</tag >的方式开始和关闭标签，而是通过<… />标签直接开始和结束标签。换行标签(< br >)就是一种空元素。正确关闭空元素的方法是在开始标签末尾直接添加斜杠，例如< br/>。目前，HTML、XHTML 和 XML 都接受这种方式。也就是说，即使< br >在所有浏览器中都是有效的，但是使用< br/>其实是更有保障的做法。

(3) 标签大小写：目前，HTML 标签不区分大小写，例如< body >和< BODY >代表相同的意思，虽然有许多网站使用大写的 HTML 标签。值得注意的是，万维网联盟(W3C)在 HTML4 中推荐使用小写，而在 XHTML 版本中强制使用小写。

从上述内容中可以看出，在当前的互联网环境下，HTML 文档的编写并没有严格的要求和组织，存在一些标签的缺失和不规范，这就需要使用工具去补全 HTML 文档的结构。表 6.1 中列举了一些最常用的 HTML 元素。

**表 6.1　一些常用的 HTML 元素**

| 元　　素 | 描　　述 |
|---|---|
| <! --...--> | 定义注释 |
| <! DOCTYPE > | 定义文档类型 |
| < a > | 超链接 |
| < address > | 定义文档作者或拥有者的联系信息 |
| < aside > | 定义页面内容之外的内容 |
| < audio > | 定义声音内容 |
| < base > | 定义页面中所有链接的默认地址或默认目标 |
| < body > | 定义文档的主体 |
| < br > | 定义简单的折行 |
| < div > | 定义文档中的节 |
| < dl > | 定义列表 |
| < form > | 定义供用户输入的 HTML 表单 |
| < frame > | 定义框架集的窗口或框架 |
| < h1 > ～ < h6 > | 定义 HTML 标题 |
| < head > | 定义关于文档的信息 |
| < hr > | 定义水平线 |
| < html > | 定义 HTML 文档 |
| < iframe > | 定义内联框架 |
| < img > | 定义图像 |
| < object > | 定义内嵌对象 |

续表

| 元　　素 | 描　　述 |
|---|---|
| <ol> | 定义有序列表 |
| <p> | 定义段落 |
| <script> | 定义客户端脚本 |
| <span> | 定义文档中的节 |
| <style> | 定义文档的样式信息 |
| <table> | 定义表格 |
| <title> | 定义文档的标题 |
| <ul> | 定义无序列表 |
| <var> | 定义文本的变量部分 |
| <video> | 定义视频 |

### 6.1.3 HTML 属性

HTML 标签是可以设置并拥有属性的，属性提供了关于 HTML 元素的附加信息，在 HTML 元素中属性一般出现在开始标签中，并且总是以名称/值对的形式出现，例如 name="value"。

下面举几个 HTML 属性实例。

属性例子 1：

<h1>定义标题的开始。

<h1 align="center">拥有关于对齐方式的附加信息，表示居中排列标题。

属性例子 2：

<body>定义 HTML 文档的主体。

<body bgcolor="yellow">拥有关于背景颜色的附加信息，表示将背景颜色设置为黄色。

属性有以下注意事项。

(1) 大小写：属性和属性值不区分字母大小写，不过万维网联盟在其 HTML 推荐标准中推荐使用小写的属性名/属性值，在 XHTML 中要求使用小写属性。

(2) 值应该包含在引号内：属性值应该始终被包括在引号内，双引号是最常用的，不过使用单引号也没有问题。在某些个别的情况下，比如属性值本身就含有双引号的情况，必须使用单引号，例如 name='Bill "HelloWorld" Gates'。

## 6.2　XML

XML 指可扩展标记语言(Extensible Markup Language),它是一种用于标记电子文件使其具有结构性的标记语言,类似于 HTML。1998 年 2 月,W3C 正式批准了可扩展标记语言的标准定义,是 W3C 的推荐标准。XML 可对文档和数据进行结构化处理,从而能够在企业内、外部进行数据交换,实现动态内容的生成。XML 可帮助人们更加准确地搜索、更加方便地传送软件组件、更好地描述一些事物。

XML 是各种应用程序之间进行数据传输的最常用的工具。XML 的设计宗旨是传输数据,而不是显示数据。此外,XML 标签没有被预定义,用户需要根据实际需求自行定义标签。XML 文档不会对标签或数据内容本身做任何变换,它只是被设计用来结构化、存储以及传输信息。用户需要通过编写程序或软件才能传送、接收和显示 XML 文档。

HTML 和 XML 都是标准通用标记语言的子集,但 XML 不是 HTML 的替代,XML 和 HTML 是为不同目的设计的:

(1) XML 被设计用来传输和存储数据,其焦点是数据的内容。

(2) HTML 被设计用来显示数据,其焦点是数据的外观。

简单地说,HTML 旨在显示信息,而 XML 旨在传输信息。此外,与 HTML 相比,XML 的标记需要成对出现,并且字母区分大小写。另外,存在错误的 HTML 文档是可以编译的,但是存在语法错误的 XML 文档应该避免进行编译。

目前,XML 在 Web 中起到的作用已经完全不亚于一直作为 Web 基石的 HTML。XML 正成为各种应用程序之间进行数据传输的最常用工具,并且在信息存储和描述领域变得越来越流行。

XML 具有以下用途。

(1) 实现 HTML 布局与数据的分离:XML 文件可以独立地存储数据,因此用户可专注于 HTML 的布局和显示,而无须因数据的变更修改 HTML 文档。

(2) 简化数据共享:作为一种通用标记语言,XML 可以在不同的计算机系统、不同的操作系统、不同的应用程序之间交换数据,从而使数据的共享更加简单。

(3) 简化数据传输:XML 使不兼容系统之间的数据传输更加轻松。

(4) 简化平台变更：由于 XML 在数据共享和传输方面的功能，平台的变更更加自由。

(5) 创建新的 Internet 语言：例如 XHML、WAP、WAML、RSS 等互联网常用技术都是通过 XML 创建的。

在论述了 XML 的特点和用途之后，下面把重点转移到其本身——结构和语法。

### 6.2.1 XML 的结构和语法

XML 文档形成了一种树结构，它从“根部”开始，然后扩展到“枝叶”。XML 文档必须包含根元素，该元素是其他所有元素的父元素。一个形象的描述是，XML 文档中的元素形成了一棵文档树，这棵树从根部开始，并一直扩展到树的最底端，所有元素均可拥有子元素。父、子以及同胞等术语用于描述元素之间的关系。父元素拥有子元素，相同层级上的子元素称为同胞，所有元素均可拥有文本内容和属性。

下面用一个具体的例子来讲述 XML，其代码表示一本书。

**【例 6-1】** XML 详解。

```
<bookstore>
<book category="COOKING">
  <title lang="en">Everyday Italian</title>
  <author>Giada De Laurentiis</author>
  <year>2005</year>
  <price>30.00</price>
</book>
<book category="CHILDREN">
  <title lang="en">Harry Potter</title>
  <author>J K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
</book>
<book category="WEB">
  <title lang="en">Learning XML</title>
  <author>Erik T. Ray</author>
  <year>2003</year>
  <price>39.95</price>
</book>
</bookstore>
```

该例中的根元素是<bookstore>，文档中的所有<book>元素被包含在<bookstore>中。<book>元素有 4 个子元素，即<title>、<author>、<year>、<price>，而且<title>、

< author >、< year >、< price >互为同胞元素。

XML 的语法非常简单、清晰，一个合法的 XML 文档应具有以下特点。

(1) 元素必须要有关闭标签，而 HTML 在某些情况下可以省略关闭标签。

(2) 标签区分字母大小写，而 HTML 不区分字母大小写。在 XML 中，标签< Letter >与标签< letter >是不同的。

(3) 必须正确地嵌套：在 HTML 中用户经常会看到没有正确嵌套的元素，例如< b >< i >This text is bold and italic </b ></i >；在 XML 中所有元素都必须彼此正确地嵌套，例如< b >< i > This text is bold and italic </i ></b >。在这里正确嵌套的意思是，由于< i >元素是在< b >元素内打开的，那么它必须在< b >元素内关闭。

(4) 文档必须要有根元素，且只能有一个根元素，即必须要有一个元素是其他所有元素的父元素。例如：

**【例 6-2】** 根元素。

```
<root>
  <child>
    <subchild>…</subchild>
  </child>
</root>
```

(5) 属性值需要添加引号：与 HTML 类似，XML 也可拥有属性(名称/值对)。其中，属性值需要用单引号或者双引号括起来。请研究下面两个 XML 文档，第一个是错误的，第二个是正确的。

**【例 6-3】** 格式对比。

```
<note date = 08/08/2008 >
<to> George </to>
<from> John </from>
</note>

<note date = "08/08/2008">
<to> George </to>
<from> John </from>
</note>
```

第一个文档中的错误是，< note >元素中 date 属性的值没有加引号。

(6) 实体引用：在 XML 中一些字符有特殊的意义。如果用户把字符“<”放

在 XML 元素中，会发生错误，这是因为解析器会把它当作新元素的开始，这样会产生 XML 错误，例如< message > if salary < 1000 then </message >。为了避免产生这个错误，请用实体引用来代替“<”字符，即< message > if salary < 1000 then </message >。

在 XML 中有 5 个预定义的实体引用，如表 6.2 所示。

**表 6.2 实体引用介绍**

| 替代符号 | 原 符 号 | 含 义 |
|---|---|---|
| < | < | 小于 |
| > | > | 大于 |
| & | & | 和号 |
| ' | ' | 单引号 |
| " | " | 引号 |

与 HTML 不同，XML 文本内容中的空格会被保留。HTML 会把多个连续的空格字符裁减（合并）为一个。

```
HTML: Hello           my name is David.
输出: Hello my name is David.
```

在 XML 中，文档中的空格不会被删节。

(7) XML 以 LF 存储换行：在 Windows 应用程序中，换行通常以一对字符来存储，即回车符（CR）和换行符（LF）。这对字符与打字机设置新行的动作有相似之处。在 UNIX 应用程序中，新行以 LF 字符存储，而 Macintosh 应用程序使用 CR 存储新行。

### 6.2.2 XML 元素和属性

和 HTML 一样，XML 也由元素构成。XML 元素指的是从（且包括）开始标签直到（且包括）结束标签的所有部分。元素可以包含其他元素、文本或者两者的混合物，元素也可以拥有属性。与 HTML 不同的是，XML 的元素标签都是用户自定义的，而非预定义的。因此，XML 中的标签可以有任意多个，并且可以表达一定的实际含义，这也是 XML 可作为数据传输和存储文档的关键所在。其次，XML 元素是可扩展的，这也使得它可以携带更多的信息。请看下面这个 XML 例子：

【例 6-4】 XML 扩展。

```
<note>
<to>George</to>
<from>John</from>
<body>Don't forget the meeting!</body>
</note>
```

假设创建了一个应用程序，可将< to >、< from >以及< body >元素提取出来，并产生以下输出：

```
MESSAGE
To:     George
From:   John

Don't forget the meeting!
```

之后又向这个文档添加了一些额外的信息：

```
<note>
<date>2008 - 08 - 08</date>
<to>George</to>
<from>John</from>
<heading>Reminder</heading>
<body>Don't forget the meeting!</body>
</note>
```

想一下，这个应用程序会中断或崩溃吗？

答案是不会。这个应用程序仍然可以找到 XML 文档中的< to >、< from >以及< body>元素，并产生同样的输出。也就是说，即使在一些编辑完成的 XML 文档中插入新内容也不会影响到其他应用程序对原有文档数据的提取。XML 的优势之一是可以经常在不中断应用程序的情况下进行扩展。

XML 的元素大部分是用户自定义的，因此其命名需要遵循一定的规则：

(1) 名称可以包含字母、数字以及其他字符。

(2) 名称不能以数字或者标点符号开始。

(3) 名称不能以字符“xml”(或者 XML、Xml)开始。

(4) 名称不能包含空格。

因为 XML 没有保留字，所以在遵循以上规则的前提下可以使用任意字符作为元素的名称。但是，由于 XML 是用于数据传输的，元素的命名最好可标识相应的实际含

义，元素的名称也应该尽量简短。此外，由于 XML 中数据的提取是在其他软件中进行的，所以在命名元素时需要考虑对应软件的处理习惯，以免造成一些不必要的麻烦。

与 HTML 类似，XML 元素也可以在开始标签中包含属性，并通过属性提供有关元素的额外信息。值必须被引号包围，不过单引号和双引号均可使用。例如在 person 标签中提供一个人的性别，可以这样写：

```
<person sex="female">
```

也可以这样写：

```
<person sex='female'>
```

如果属性值本身包含双引号，那么有必要使用单引号括住它，就像这个例子：

```
<gangster name='George "Shotgun" Ziegler'>
```

或者使用实体引用：

```
<gangster name="George &quot;Shotgun&quot; Ziegler">
```

从原则上说，任何元素的开始标签都可以包含属性。但是，从 XML 的数据传输与数据存储的功能出发，会因使用属性而引起一些问题：

(1) 属性无法包含多重的值(元素可以)。

(2) 属性无法描述树结构(元素可以)。

(3) 属性不易扩展(为未来的变化)。

(4) 属性难以阅读和维护，请尽量选择用元素来描述数据，而使用属性提供与数据无关的信息。

有时候会向元素分配 ID 索引，这些 ID 索引可用于标识 XML 元素，它起作用的方式与 HTML 中的 ID 属性是一样的。下面这个例子演示了这种情况：

**【例 6-5】** ID 索引。

```
<messages>
  <note id="501">
    <to>George</to>
    <from>John</from>
```

```
        <heading>Reminder</heading>
        <body>Don't forget the meeting!</body>
    </note>
    <note id="502">
        <to>John</to>
        <from>George</from>
        <heading>Re: Reminder</heading>
        <body>I will not</body>
    </note>
</messages>
```

上面的 ID 仅仅是一个标识符,用于标识不同的便签,它并不是便签数据的组成部分。

因此读者应该有这样一个概念:元数据(有关数据的数据)应当存储为属性,而数据本身应当存储为元素。

## 6.3　利用 urllib 处理 HTTP


视频讲解

如何获取网页数据呢?简单地说,就是通过 URL 来获取 HTML 文档。在浏览器中呈现给用户的可能是排版精良、图文并茂的一个网页,其实这就是浏览器解释的结果。从根本上讲,它是一段结合了 JavaScript 和 CSS 的 HTML 代码。形象地说,如果把网页比作一个人,那么 HTML 便是骨架,JavaScript 就是肌肉,CSS 则是衣服。

本节讲解如何使用 Python 标准库中的 urllib 获取网页的 HTML 源代码,以及如何使用 BeautifulSoup4 从 HTML 中提取各种元素。

互联网中最基本的传输单元是网页。WWW 的工作基于 B/S 计算模型,由网络浏览器和网络服务器构成,两者之间采用超文本传送协议(HTTP)通信。HTTP(HyperText Transfer Protocol,超文本传输协议)构建于 TCP/IP 协议之上,是网络浏览器和网络服务器之间的应用层协议,是一种通用的、无状态的、面向对象的协议。它允许将超文本标记语言(HTML)文档从 Web 服务器传送到客户端的浏览器。除了保证计算机正确、快速地传输 HTML 以外,HTTP 还确定传输文档中的哪一部分,以及哪部分内容首先显示(例如文本先于图形)等。

一次 HTTP 操作称为一个事务,其工作过程可分为以下 4 步。

(1) 建立连接(connect):首先客户机与服务器需要建立连接,只要单击某个超链接,HTTP 的工作便开始。

(2) 浏览器请求(request):在建立连接后,客户机发送一个请求给服务器,请求方式的格式为"统一资源标识符(URL)+协议版本号+MIME信息(包括请求修饰符、客户机信息和可能的内容)"。

(3) 服务器应答(response):服务器接到请求后给予相应的响应信息,其格式为"状态行+通用信息头+响应头+实体类+报文主体"。

(4) 关闭连接(close):客户端接收服务器返回的信息,通过浏览器显示在用户的显示屏上,然后客户机与服务器断开连接。

如果以上过程中的某一步出现错误,那么产生错误的信息将返回到客户端,由显示屏输出。对于用户来说,这些过程是由HTTP完成的,用户只要用鼠标单击,等待信息显示就可以了。

下面举一个例子来展示这个典型的HTTP操作过程。

首先在浏览器上输入"http://www.maketop.net/resource/rs_041112_02.php",将浏览器连接到www.maketop.net然后发送:

**【例6-6】** HTTP操作。

```
>> GET /resource/rs_041112_02.php HTTP1.1
>> Host: www.maketop.net
>> Accept: image/gif,image/x-xbitmap,image/jpeg,image/pjpeg
>> Accept-Language: en
>> Accept-Encoding: gzip,deflate
>> User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; rv: 1.7.3) Gecko/20040913
Firefox/0.10
>> Connection: Keep-Alive
>>
```

解释:浏览器使用HTTP1.1协议请求页面"/resource/rs_041112_02.php",并告诉服务器用户的浏览器是Firefox 0.10、操作系统是Windows XP。浏览器希望保持与www.maketop.net的连接,并请求获得更多的文件,包括网页中的图片。

翻译如下:

```
>> 用 HTTP1.1 协议获得 "/resource/rs_041112_02.php"
>> 访问的主机: www.maketop.net
>> 接收的文件: image/gif、image/x-xbitmap、image/jpeg、image/pjpeg
>> 使用的语言: en
>> 接收的编码方式(浏览器能够解释的): gzip、deflate
>> 用户的浏览器信息: Windows XP 的操作系统、Firefox 0.10 的浏览器
>> 保持连接: 还要取图片
>>
```

www.maketop.net 的服务器发出响应：

```
<< HTTP/1.1 200 OK
<< Date: Mon,12 Mar 2004 19:12:16 GMT
<< Server: Apache/1.3.31 (UNIX) mod_throttle/3.1.2
<< Last-Modified: Fri,22 Sep 2004 14:16:18
<< ETag: "dd7b6e-d29-39cb69b2"
<< Accept-Ranges: bytes
<< Content-Length: 3369
<< Connection: close
<< Content-Type: text/html
<<
<< File content goes here
```

浏览器从服务器的响应中获得服务器的信息，例如运行在 Apache。

上面的语言翻译如下：

```
<< HTTP1.1 协议方式有效
<< 当前时间: Mon,12 Mar 2004 19:12:16 GMT
<< 服务器: Apache/1.3.31 (UNIX) mod_throttle/3.1.2
<< 最后一次修改: Fri,22 Sep 2004 14:16:18
<< ETag: "dd7b6e-d29-39cb69b2"
<< Accept-Ranges: bytes
<< Content-Length: 3369
<< Connection: close
<< Content-Type: text/html
<<
<< File content goes here
```

上面的例子就是最简单的交互过程描述。

urllib 提供了一系列用于操作 URL 且进一步获取 URL 所定位数据文档的高层接口。其中，该模块的 urlopen()方法类似于 Python 的内置方法 open()，但以 url 作为参数。此外，urllib 仅支持以只读方式打开 url，并且没有类似 seek()的方法定义指针。

urllib.request.urlopen(url[,data[,proixes,[,context]]])用于打开一个由 url 标记的网络对象，以进行读取。第 1 个参数 url 即为 URL，第 2～4 个参数可以不传递。第 2 个参数 data 可用于传递某个 POST 请求(通常请求类型为 GET)，该参数值最好使用 urlencode()方法获取。第 3 个参数为 proxies。urlopen()函数对代理的使用非常透明，并不要求身份验证。在 UNIX 或者 Windows 环境下进行 Python 编译之前，需要为 URL 设置 http_proxy 和 ftp_proxy 环境变量，以定位到目标代理服务器。proxies 参数应该设定为一个主机后缀的逗号分隔列表，可选择在 URL 后附加

“:port”。在 Windows 环境下，如果未设置代理环境变量，则代理设置参数采用注册表中的 Internet 设置；在 Mac OS X 环境下，代理设置采用 OS X 系统配置框架。当然，用户也可以设置为不使用代理。下面是一些使用 HTTP 代理的情况及例子。

使用“http://tianqi.so.com/weather/101210111”作为 HTTP 代理：

```
>>> proxies = {'http': 'http://tianqi.so.com/weather/101210111'}
>>> filehandle = urllib.request.urlopen(some_url,proxies = proxies)
```

不使用 HTTP 代理：

```
>>> filehandle = urllib.request.urlopen(some_url,proxies = {})
```

使用系统环境中的代理设置：

```
>>> filehandle = urllib.request.urlopen(some_url,proxies = None)
>>> filehandle = urllib.request.urlopen(some_url)
```

以下是使用 GET 和 POST 方法返回网络对象的示例：

**【例 6-7】** GET 方法。

```
>>> import urllib
>>> params = urllib.parse.urlencode({'spam': 1,'eggs': 2,'bacon': 0})
>>> f = urllib.requst.urlopen("http://www.musi - cal.com/cgi - bin/query? % s" % params)
>>> print(f.read())
```

**【例 6-8】** POST 方法(代码 6-8.py)。

```
>>> import urllib
>>> params = urllib.parse.urlencode({'spam': 1,'eggs': 2,'bacon': 0})
>>> f = urllib.request.urlopen("http://www.musi - cal.com/cgi - bin/query",params)
>>> print(f.read())
```

此外，urllib 还有许多其他使用方法。

- urllib.request.urlretrieve(url[,filename[,reportbook[,data]]])：用于将一个网络对象复制到本地文件夹(或缓存)。不过，如果 url 参数指向本地文件或者一个当前对象的有效缓存设备，则这个对象不会被复制。该方法返回一个元组(filename,headers)，其中 filename 是指本地文件名，headers 则保存了网络对象的 info()方法的返回值。

**【例 6-9】** urllib. request. urlretrieve(url[,filename[,reportbook[,data]]])的使用方法。

```
>>> filename = urllib.request.urlretrieve('http://cuiqingcai.com/947.html',filename = r'C:
/1.html')
>>> type(filename)
tuple
>>> filename[0]
'C:/1.html'
>>> filename[1]
< httplib.HTTPMessage instance at 0x0000000004082688 >
>>> print(filename[1])
Server: nginx/1.4.6 (Ubuntu)
Data: Sat,16 Jul 2016 13:39:18GMT
Content - Type: text/html;charset = UTF - 8
Connection: close
X - Powered - By: PHP/5.5.9 - lubuntu4.14
Vary: Accept - Encoding,Cookie
Cache - Control: max - age = 3,must - revalidate
WP - Super - Cache: Served supercache file from PHP
```

- urllib. request. urlcleanup()：用于清除之前引用 urlretrieve()方法产生的缓存。
- urllib. parse. quote(string[,safe])：用%xx 代替字符串中的一些特殊字符。

**【例 6-10】** urllib. parse. quote(string[,safe])的使用方法。

```
>>> urllib.parse.quote('http://www.cnblogs.com/sysu - blackbear/p/3629420.html')
'http%3A//www.cnblog.com/sysu - blackbear/p/3629420.html'
#使用%3A代替冒号
```

- urllib. parse. quote_plus(string[,safe])：和 urllib. quote(string[,safe])类似，不过字符串中的空格使用＋代替。

**【例 6-11】** urllib. parse. quote_plus(string[,safe])的使用方法。

```
>>> urllib.parse.quote_plus('http://www.cnblogs.com/sysu - blackbear/p/3629 420.html')
'http%3A//www.cnblogs.com/sysu - blackbear/p/3629 + 420.html'
#使用%3A代替冒号,并使用+代替空格
```

- urllib. parse. unquote(string)：urllib. quote()的逆操作。

**【例 6-12】** urllib. parse. unquote(string)的使用方法。

```
>>> urllib.parse.unquote('http%3A//www.cnblogs.com/sysu - blackbear/p/3629420.html')
'http://www.cnblogs.com/sysu - blackbear/p/3629420.html'
```

- urllib. parse. unquote_plus(string)：urllib. quote_plus()的逆操作。

【例 6-13】 urllib. parse. unquote_plus(string)的使用方法。

```
>>> urllib.parse.unquote_plus('http%3A//www.cnblogs.com/sysu-blackbear/p/3629 + 420.html')
'http://www.cnblogs.com/sysu-blackbear/p/3629 420.html'
```

- urllib. parse. urlencode(query[,doseq])：用于将一个映射对象或者一个两元素元组序列转化为一个由%编码的字符串，传递给 urlopen()作为可选声明 data 的值。

【例 6-14】 urllib. parse. urlencode(query[,doseq])的使用方法(代码 6-14. py)。

```
>>> params = urllib.parse.urlencode({'egg':1,'fruit':2,'bird':3})
>>> params
'egg=1&fruit=2&bird=3'
```

到目前为止，读者可能已经发现，urllib 把 HTTP 协议的 3 个步骤(建立连接、发出请求和收到响应)统一在 urlopen()方法中实现。然而，在很多情况下这并不能保证可以顺利获取目标 URL 对应的数据文件。现在大多数网站都是动态网页，在获取网页的过程中需要动态地传递参数，它再对此做出相应的响应。所以，在访问一些网页时需要传递数据。

Python 标准库中的另一个 URL 处理包 urllib2 可以构建一个 Request 类的实例来设置 URL 请求的 Headers，因此可通过 urllib 模块伪装浏览器，进而能处理更复杂的 HTTP 访问。当然，urllib2 不能代替 urllib，例如 urllib 并没有提供 urlencode()方法用来生成 GET 查询字符串，且 urllib. urlretrieve()函数以及 urllib. quote()等一系列 quote()和 unquote()方法都没有加入到 urllib2。这也是在实际应用中一起使用 urllib 和 urllib2 的原因。

## 6.4 利用 BeautifulSoup4 解析 HTML 文档

在利用 urllib 获取目标 HTML 文档之后，接下来就要对文档中的内容进行析取。关键问题在于，HTML 文档中有很大一部分内容是用于设置文档呈现形式的，而这部分内容在很多情况下并不被用户关心。用户关心更多的可能是网页正文内

容中的某些信息，或者网页内的超链接。用户可以使用 Python 标准库中的 re 模块，通过构建模式对象的方式析取出满足用户需求的文本。然而，在实践中并不推荐用户使用这种方法。re 模式的构建较为复杂，且构建好的模式难以推广到多个案例中。

Python 的第三方库 BeautifulSoup 在处理 HTML 和 XML 编码文档方面的表现非常优秀。BeautifulSoup 是一个可以从 HTML 或 XML 文件中提取数据的 Python 库，它能够通过用户喜欢的转换器实现惯用的文档导航、查找、修改文档等功能。BeautifulSoup 模块可以很好地处理不规范标记并生成剖析树，且提供简单又常用的导航、搜索以及修改剖析树的操作，特别是 BeautifulSoup 的一些关键函数可以结合正则表达式 re 模块中的模式或者使用 CSS 查询器语法，在很大程度上减少了用户花在编程上的时间。

BeautifulSoup 将复杂的 HTML 或 XML 转化成树形结构，每个结点都是 Python 对象。这里借用官方文档中的一个例子来对 BeautifulSoup4 的属性和方法进行演示。这是《爱丽丝梦游仙境》中的一段内容。

```
html_doc = """
<html><head><title>The Dormouse's story</title></head>
<body>
<p class = "title"><b>The Dormouse's story</b></p>

<p class = "story">Once upon a time there were three little sisters; and their names were
<a href = "http://example.com/elsie" class = "sister" id = "link1">Elsie</a>,
<a href = "http://example.com/lacie" class = "sister" id = "link2">Lacie</a> and
<a href = "http://example.com/tillie" class = "sister" id = "link3">Tillie</a>;
and they lived at the bottom of a well.</p>

<p class = "story">...</p>
"""
```

使用 BeautifulSoup 解析这段代码能够得到一个 BeautifulSoup 的对象，并能按照标准缩进格式的结构输出：

**【例 6-15】** BeautifulSoup 模块的使用。

```
>>> from bs4 import BeautifulSoup
>>> soup = BeautifulSoup(html_doc, 'html.parser')
>>> print(soup.prettify())
<html>
```

```
<head>
  <title>
   The Dormouse's story
  </title>
</head>
<body>
  <p class="title">
   <b>
    The Dormouse's story
   </b>
  </p>
  <p class="story">
   Once upon a time there were three little sisters; and their names were
   <a class="sister" href="http://example.com/elsie" id="link1">
    Elsie
   </a>
   <a class="sister" href="http://example.com/lacie" id="link2">
    Lacie
   </a>
   and
   <a class="sister" href="http://example.com/tillie" id="link2">
    Tillie
   </a>
   ; and they lived at the bottom of a well.
  </p>
  <p class="story">
   ...
  </p>
</body>
</html>
```

### 6.4.1 BeautifulSoup4 中的对象


视频讲解

BeautifulSoup 将复杂的 HTML 文档转换成一个复杂的树形结构，每个结点都是 Python 对象，所有对象可以归纳为 4 种，即 Tag、NavigableString、BeautifulSoup 和 Comment。

#### 1. Tag 对象

BeautifulSoup 中的 Tag 对象与 XML 或 HTML 原生文档中的 Tag 相同。Tag 对象通过.Tag 的方式获取对应类别的第一个标签对象：

【例 6-16】 Tag 对象的介绍。

```
>>> soup = BeautifulSoup('<b class = "boldest"> Extremely bold </b>')
>>> tag = soup.b
>>> type(tag)
<class 'bs4.element.Tag'>
```

Tag 有很多方法和属性，现在介绍一下其最重要的属性 name 和 attributes。首先讲解 name 属性。

每个 Tag 都有自己的名字，通过.name 来获取：

```
>>> tag.name
'b'
```

如果改变了 Tag 的 name，将影响所有通过当前 BeautifulSoup 对象生成的 HTML 文档：

```
>>> tag.name = "blockquote"
>>> tag
<blockquote class = "boldest"> Extremely bold </blockquote>
```

接下来讲解 attributes 属性。一个 Tag 可能有很多个属性。在 tag <b class="boldest">中有一个“class”属性，值为“boldest”。Tag 属性的操作方法与字典相同：

```
>>> tag['class']
['bddest']
```

用户也可以直接“点”取属性，例如“.attrs”：

```
>>> tag.attrs
{'class': ['boldest']}
```

Tag 的属性可以被添加、删除或修改。再说一次，Tag 属性的操作方法与字典一样。

```
>>> tag['class'] = 'verybold'
>>> tag['id'] = 1
>>> tag
```

```
<blockquote class = "verybold" id = "1"> Extremely bold </blockquote>
>>> del tag['class']
>>> del tag['id']
>>> tag
<blockquote> Extremely bold </blockquote>
```

如果引用了 Tag 中一个不存在的属性，则返回 None：

```
>>> tag['class']
KeyError: 'class'
>>> print(tag.get('class'))
None
```

**2. NavigableString 对象**

接下来可通过.string 获取标签中的文本内容，文本类型为 NavigableString。字符串常被包含在 Tag 内。BeautifulSoup 用 NavigableString 类来包装 Tag 中的字符串：

**【例 6-17】** NavigableString 对象的介绍。

```
>>> tag.string
Extremely bold
>>> type(tag.string)
<class 'bs4.element.NavigableString'>
```

Tag 中包含的字符串不能编辑，但是可以被替换成其他字符串，用 replace_with() 方法：

```
>>> tag.string.replace_with("No longer bold")
>>> tag
<blockquote> No longer bold </blockquote>
```

**3. BeautifulSoup 对象**

BeautifulSoup 对象表示的是一个文档的全部内容，在大部分时候，可以把它当作 Tag 对象，它支持遍历文档树和搜索文档树中描述的大部分方法。

因为 BeautifulSoup 对象并不是真正的 HTML 或 XML 的 Tag，所以它没有 name 和 attribute 属性。但有时查看它的.name 属性是很方便的，所以 BeautifulSoup

对象包含了一个值为“[document]”的特殊属性.name：

【例 6-18】 BeautifulSoup 对象的介绍。

```
>>> soup.name
[document]
```

#### 4. Comment 对象

Tag、NavigableString、BeautifulSoup 几乎覆盖了 HTML 和 XML 中的所有内容，但还有一些特殊对象容易让人担心内容是文档的注释部分。Comment 对象是一种特殊类型的 NavigableString 对象。

【例 6-19】 Comment 对象的介绍。

```
>>> markup = "<b><!--Hey, buddy. Want to buy a used parser?--></b>"
>>> soup = BeautifulSoup(markup)
>>> comment = soup.b.string
>>> type(comment)
<class 'bs4.element.Comment'>
```

当它出现在 HTML 文档中时，Comment 对象会使用特殊的格式输出：

```
>>> print(soup.b.prettify())
<b>
<!--Hey, buddy. Want to buy a used parser?-->
</b>
```

### 6.4.2 遍历文档树

视频讲解

一个标签可能包含许多字符串或者其他标签，从文档树的视角看，这些都是该标签的子结点。通过 Tag 方法可以获取标签对象的子结点标签，并且可在一个语句中多次使用。通过.string 方法可获取标签对象的字符串子结点。BeautifulSoup 提供了许多操作和遍历子结点的属性。注意，BeautifulSoup 中的字符串结点不支持这些属性，因为字符串没有子结点。

这里仍用《爱丽丝梦游仙境》文档来做例子：

【例 6-20】 遍历文档树示例。

```
html_doc = """
<html><head><title>The Dormouse's story</title></head>
```

```
    <body>
<p class="title"><b>The Dormouse's story</b></p>
<p class="story">Once upon a time there were three little sisters; and their names were
<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>,
<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and
<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;
and they lived at the bottom of a well.</p>
<p class="story">...</p>
"""

>>> from bs4 import BeautifulSoup
>>> soup = BeautifulSoup(html_doc, 'html.parser')
```

通过这个例子演示了怎样从文档的一段内容找到另一段内容。

操作文档树最简单的方法就是告诉它想获取的 Tag 的 name。如果想获取<head>标签，只要用 soup.head：

```
>>> soup.head
<head><title>The Dormouse's story</title></head>
>>> soup.title
<title>The Dormouse's story</title>
```

下面的代码可以获取<body>标签中的第一个<b>标签：

```
>>> soup.body.b
<b>The Dormouse's story</b>
```

通过点取属性的方式只能获得当前名字的第一个 Tag：

```
>>> soup.a
<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>
```

如果想要得到所有的<a>标签，或者通过名字得到比一个 Tag 更多的内容，则需要用到其他方法，例如 find_all()：

```
>>> soup.find_all('a')
[<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
<a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

### 1．.contents 和.children

Tag 的.contents 属性可以将 Tag 的子结点以列表的方式输出：

**【例 6-21】** .contents 和.children 的介绍。

```
>>> head_tag = soup.head
>>> head_tag
<head><title>The Dormouse's story</title></head>
>>> head_tag.contents
[<title>The Dormouse's story</title>]
>>> title_tag = head_tag.contents[0]
>>> title_tag
<title>The Dormouse's story</title>
>>> title_tag.contents
[The Dormouse's story]
```

BeautifulSoup 对象本身一定会包含子结点，也就是说<html>标签也是 BeautifulSoup 对象的子结点：

```
>>> len(soup.contents)
1
>>> soup.contents[0].name
html
```

字符串没有.contents 属性，因为字符串没有子结点：

```
>>> text = title_tag.contents[0]
>>> text.contents
AttributeError: 'NavigableString' object has no attribute 'contents'
```

通过 Tag 的.children 生成器可以对 Tag 的子结点进行循环：

```
>>> for child in title_tag.children:
print(child)
The Dormouse's story
```

### 2．.descendants

.contents 和.children 属性仅包含 Tag 的直接子结点。例如，<head>标签只有一个直接子结点<title>：

【例 6-22】 .descendants 的介绍。

```
>>> head_tag.contents
[<title>The Dormouse's story</title>]
```

但是< title >标签也包含一个子结点——字符串"The Dormouse's story",在这种情况下字符串"The Dormouse's story"属于< head >标签的子孙结点。.descendants 属性可以对所有 Tag 的子孙结点进行递归循环：

```
>>> for child in head_tag.descendants:
        print(child)
<title>The Dormouse's story</title>
The Dormouse's story
```

在上面的例子中，< head >标签只有一个子结点，但是有两个子孙结点——< head >结点和< head >的子结点，BeautifulSoup 有一个直接子结点(< html >结点)，却有很多子孙结点：

```
>>> len(list(soup.children))
1
>>> len(list(soup.descendants))
25
```

**3. .string**

如果 Tag 只有一个 NavigableString 类型的子结点，那么这个 Tag 可以使用.string 得到子结点：

【例 6-23】 .string 的介绍。

```
>>> title_tag.string
The Dormouse's story
```

如果一个 Tag 仅有一个子结点，那么这个 Tag 也可以使用.string 方法，输出结果与当前唯一子结点的.string 结果相同：

```
>>> head_tag.contents
[<title>The Dormouse's story</title>]
>>> head_tag.string
The Dormouse's story
```

如果 Tag 包含了多个子结点，Tag 将无法确定. string 方法应该调用哪个子结点的内容，. string 的输出结果是 None：

```
>>> print(soup.html.string)
None
```

#### 4. . strings 和. stripped_strings

如果 Tag 中包含多个字符串，可以使用. strings 循环获取：

**【例 6-24】** . strings 和. stripped_strings 示例。

```
>>> for string in soup.strings:
        print(repr(string))
    The Dormouse's story
    '\n\n'
    The Dormouse's story
    '\n\n'
    'Once upon a time there were three little sisters; and their names were\n'
    'Elsie'
    ',\n'
    'Lacie'
    ' and\n'
    'Tillie'
    ';\nand they lived at the bottom of a well.'
    '\n\n'
    '...'
    '\n'
```

在输出的字符串中可能包含了很多空格或空行，使用. stripped_strings 可以去除多余的空白内容：

```
>>> for string in soup.stripped_strings:
        print(repr(string))
    The Dormouse's story
    The Dormouse's story
    'Once upon a time there were three little sisters; and their names were'
    'Elsie'
    ','
    'Lacie'
    'and'
    'Tillie'
    ';\nand they lived at the bottom of a well.'
    '...'
```

这样，全部是空格的行会被忽略掉，段首和段末的空白会被删除。

上面介绍了如何用 BeautifulSoup 对象和 Tag 对象的属性遍历文档树，但这只是一部分，更多可用属性见表 6.3。

**表 6.3 文档搜索属性**

| 属　性 | 描　述 |
| --- | --- |
| .contents | 可以将 Tag 的子结点以列表的方式输出 |
| .children | 生成器，可以对 Tag 的子结点进行循环 |
| .descendants | 可以对所有 Tag 的子孙结点进行递归循环 |
| .strings | 如果 Tag 中包含多个字符串，可以使用.strings 循环获取 |
| .stripped_strings | 输出的字符串中可能包含了很多空格或空行，使用.stripped_strings 可以去除多余的空白内容 |
| .parent | 获取某个元素的父结点 |
| .parents | 可以递归得到元素的所有父辈结点 |
| .next_sibling | 查询下一个兄弟结点；如果没有，则返回 None |
| .previous_sibling | 查询上一个兄弟结点；如果没有，则返回 None |
| .next_siblings | 向后迭代当前结点的兄弟结点 |
| .previous_siblings | 向前迭代当前结点的兄弟结点 |
| .next_element | 指向解析过程中下一个被解析的对象 |
| .previous_element | 指向当前被解析的对象的前一个解析对象 |
| .next_elements | 通过.next_elements 的迭代器就可以向后访问文档的解析内容 |
| .previous_elements | 通过.previous_elements 的迭代器就可以向前访问文档的解析内容 |

### 6.4.3 搜索文档树


视频讲解

BeautifulSoup 定义了很多搜索方法，这里着重介绍 find()和 find_all()方法，其他方法的参数和用法与它们类似。

这里仍以《爱丽丝梦游仙境》文档作为例子。

**【例 6-25】** 搜索文档树的介绍。

```
html_doc = """
<html><head><title>The Dormouse's story</title></head>
<body>
<p class = "title"><b>The Dormouse's story</b></p>
<p class = "story">Once upon a time there were three little sisters; and their names were
<a href = "http://example.com/elsie" class = "sister" id = "link1">Elsie</a>,
<a href = "http://example.com/lacie" class = "sister" id = "link2">Lacie</a> and
<a href = "http://example.com/tillie" class = "sister" id = "link3">Tillie</a>;
and they lived at the bottom of a well.</p>
<p class = "story">...</p>
"""
```

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(html_doc, 'html.parser')
```

使用 find_all()方法可以查找到想要查找的文档内容。

在介绍 find_all()方法之前先介绍一下过滤器的类型，这些过滤器贯穿整个搜索的 API。过滤器可以被用在 Tag 的 name 中、结点的属性中、字符串中或它们的混合中，具体见表 6.4。

**表 6.4　过滤器类型**

| 参数类型 | 描　　述 |
|---|---|
| 字符串 | 最简单的过滤器，匹配标签或文本内容，返回列表 |
| 正则表达式 | 通过正则表达式的 match()匹配标签或文本内容，返回列表 |
| 列表 | 返回所有与列表元素匹配的标签或文本内容 |
| True | 匹配标签或文本内容的任何值，返回列表 |
| 函数 | 若无合适的过滤器，定义一个只接受一个元素参数且返回逻辑值 True 或 False 的函数，进而匹配满足特定条件的标签或文本内容，返回列表 |

### 1. 字符串

最简单的过滤器是字符串。在搜索方法中传入一个字符串参数，BeautifulSoup 会查找与字符串完整匹配的内容。下面的例子用于查找文档中所有的<b>标签：

**【例 6-26】**　字符串的介绍。

```
>>> soup.find_all('b')
[<b>The Dormouse's story</b>]
```

如果传入字节码参数，BeautifulSoup 会当成 UTF-8 编码，可以通过传入一段 Unicode 编码来避免 BeautifulSoup 解析编码出错。

### 2. 正则表达式

如果传入正则表达式作为参数，BeautifulSoup 会通过正则表达式的 match()来匹配内容。在下面的例子中找出所有以“b”开头的标签，这表示<body>和<b>标签都应该被找到：

**【例 6-27】**　正则表达式的介绍。

```
>>> import re
>>> for tag in soup.find_all(re.compile("^b")):
```

```
            print(tag.name)
body
b
```

下面的代码找出名字中包含“t”的所有标签：

```
>>> for tag in soup.find_all(re.compile("t")):
            print(tag.name)
html
title
```

**3. 列表**

如果传入列表参数，BeautifulSoup 会将与列表中任一元素匹配的内容返回。下面的代码找到文档中所有的<a>标签和<b>标签：

**【例 6-28】** 列表的介绍。

```
>>> soup.find_all(["a","b"])
[<b>The Dormouse's story</b>,
<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
<a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

**4. True**

True 可以匹配任何值，下面的代码查找出所有的 Tag，但是不会返回字符串结点。

**【例 6-29】** True 的介绍。

```
>>> for tag in soup.find_all(True):
            print(tag.name)
html
head
title
body
p
b
p
a
a
a
p
```

### 5. 函数

如果没有合适的过滤器，还可以定义一个方法，方法只接受一个元素参数。如果这个方法返回 True，表示当前元素匹配并且被找到，如果不是，则返回 False。

下面的方法检验了当前元素，如果包含 class 属性却不包含 id 属性，那么将返回 True：

**【例 6-30】** 函数的介绍。

```
>>> def has_class_but_no_id(tag):
        return tag.has_attr('class') and not tag.has_attr('id')
```

将这个方法作为参数传入 find_all()方法，将得到所有<p>标签：

```
>>> soup.find_all(has_class_but_no_id)
[<p class="title"><b>The Dormouse's story</b></p>,
<p class="story">Once upon a time there were...</p>,
<p class="story">...</p>]
```

在返回结果中只有<p>标签没有<a>标签，因为<a>标签还定义了“id”；没有返回<html>和<head>，因为<html>和<head>中没有定义“class”属性。

在通过一个方法过滤一类标签属性的时候，这个方法的参数是要被过滤的属性的值，而不是这个标签。下面的例子是找出 href 属性不符合指定正则表达式的<a>标签：

```
>>> def not_lacie(href):
        return href and not re.compile("lacie").search(href)
>>> soup.find_all(href=not_lacie)
[<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
<a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

标签过滤方法可以使用复杂方法。下面的例子可以过滤出前、后都有文字的标签：

```
>>> from bs4 import NavigableString
>>> def surrounded_by_strings(tag):
        return (isinstance(tag.next_element,NavigableString)
```

```
            and isinstance(tag.previous_element,NavigableString))

>>> for tag in soup.find_all(surrounded_by_strings):
        print tag.name
p
a
a
a
p
```

下面来了解搜索方法 find_all()的细节,其格式如下:

```
find_all(name,attrs,recursive,string, ** kwargs)
```

find_all()方法搜索当前 Tag 的所有 Tag 子结点,并判断是否符合过滤器的条件。这里有几个例子:

**【例 6-31】** find_all()的介绍。

```
>>> soup.find_all("title")
[<title>The Dormouse's story</title>]
>>> soup.find_all("p","title")
[<p class="title"><b>The Dormouse's story</b></p>]
>>> soup.find_all("a")
[<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
<a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
>>> soup.find_all(id="link2")
[<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]
>>> import re
>>> soup.find(string=re.compile("sisters"))
'Once upon a time there were three little sisters; and their names were\n'
```

这里有几个方法很相似,还有几个方法是新的,参数中的 string 和 id 是什么含义?为什么 find_all("p","title")返回的是 CSS class 为"title"的< p >标签?接下来仔细看一下 find_all()的参数。

name 参数可以查找所有名字为 name 的 Tag,字符串对象会被自动忽略掉。其简单的用法如下:

```
>>> soup.find_all("title")
[<title>The Dormouse's story</title>]
```

注意，name 参数的值可以是任一类型的过滤器、字符串、正则表达式、列表、方法或者 True。

如果一个指定名字的参数不是搜索内置的参数名，在搜索时会把该参数当作指定名字 Tag 的属性来搜索，如果包含一个名字为 id 的参数，BeautifulSoup 会搜索每个 Tag 的“id”属性：

```
>>> soup.find_all(id='link2')
[<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]
```

如果传入 href 参数，BeautifulSoup 会搜索每个 Tag 的“href”属性：

```
>>> soup.find_all(href=re.compile("elsie"))
[<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>]
```

在搜索指定名字的属性时可以使用的参数值包括字符串、正则表达式、列表、True。

下面的例子在文档树中查找所有包含 id 属性的 Tag，而无论 id 的值是什么：

```
>>> soup.find_all(id=True)
[<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,
<a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

使用多个指定名字的参数可以同时过滤 Tag 的多个属性：

```
>>> soup.find_all(href=re.compile("elsie"),id='link1')
[<a class="sister" href="http://example.com/elsie" id="link1">three</a>]
```

有些 Tag 属性在搜索时不能使用，例如 HTML5 中的 data-* 属性：

```
>>> data_soup = BeautifulSoup('<div data-foo="value">foo!</div>')
>>> data_soup.find_all(data-foo="value")
SyntaxError: keyword can't be an expression
```

但是可以通过 find_all()方法的 attrs 参数定义一个字典参数来搜索包含特殊属性的 Tag：

```
>>> data_soup.find_all(attrs={"data-foo": "value"})
[<div data-foo="value">foo!</div>]
```

按照 CSS 类名搜索 Tag 的功能非常实用，但标识 CSS 类名的关键字 class 在 Python 中是保留字，使用 class 做参数会导致语法错误。从 BeautifulSoup 的 4.1.1 版本开始，可以通过 class_参数搜索有指定 CSS 类名的 Tag：

```
>>> soup.find_all("a",class_ = "sister")
[<a class = "sister" href = "http://example.com/elsie" id = "link1"> Elsie </a>,
<a class = "sister" href = "http://example.com/lacie" id = "link2"> Lacie </a>,
<a class = "sister" href = "http://example.com/tillie" id = "link3"> Tillie </a>]
```

class_参数同样接受不同类型的过滤器、字符串、正则表达式、方法或 True：

```
>>> soup.find_all(class_ = re.compile("itl"))
[<p class = "title"><b> The Dormouse's story </b></p>]
>>> def has_six_characters(css_class):
        return css_class is not None and len(css_class) == 6
soup.find_all(class_ = has_six_characters)
[<a class = "sister" href = "http://example.com/elsie" id = "link1"> Elsie </a>,
<a class = "sister" href = "http://example.com/lacie" id = "link2"> Lacie </a>,
<a class = "sister" href = "http://example.com/tillie" id = "link3"> Tillie </a>]
```

Tag 的 class 属性是多值属性。在按照 CSS 类名搜索 Tag 时，可以分别搜索 Tag 中的每个 CSS 类名：

```
>>> css_soup = BeautifulSoup('<p class = "body strikeout"></p>')
>>> css_soup.find_all("p",class_ = "strikeout")
[<p class = "body strikeout"></p>]
>>> css_soup.find_all("p",class_ = "body")
[<p class = "body strikeout"></p>]
```

在搜索 class 属性时也可以通过 CSS 值完全匹配：

```
>>> css_soup.find_all("p",class_ = "body strikeout")
[<p class = "body strikeout"></p>]
```

当完全匹配 class 的值时，如果 CSS 类名的顺序与实际不符，将搜索不到结果：

```
>>> soup.find_all("a",attrs = {"class": "sister"})
[<a class = "sister" href = "http://example.com/elsie" id = "link1"> Elsie </a>,
<a class = "sister" href = "http://example.com/lacie" id = "link2"> Lacie </a>,
<a class = "sister" href = "http://example.com/tillie" id = "link3"> Tillie </a>]
```

通过 string 参数可以搜索文档中的字符串内容。与 name 参数的可选值一样，string 参数接受字符串、正则表达式、列表、True。例如：

```
>>> soup.find_all(string = "Elsie")
[u'Elsie']
>>> soup.find_all(string = ["Tillie","Elsie","Lacie"])
[u'Elsie',u'Lacie',u'Tillie']
>>> soup.find_all(string = re.compile("Dormouse"))
[u"The Dormouse's story",u"The Dormouse's story"]
>>> def is_the_only_string_within_a_tag(s):
        ""Return True if this string is the only child of its parent tag.""
        return (s == s.parent.string)
>>> soup.find_all(string = is_the_only_string_within_a_tag)
[u"The Dormouse's story",u"The Dormouse's story",u'Elsie',u'Lacie',u'Tillie',u'...']
```

string 参数用于搜索字符串，还可以与其他参数混合使用。下面的代码用来搜索内容里面包含“Elsie”的< a >标签：

```
>>> soup.find_all("a",string = "Elsie")
[<a href = "http://example.com/elsie" class = "sister" id = "link1"> Elsie</a>]
```

find_all()方法返回全部的搜索结果，如果文档树很大，那么搜索会很慢。如果用户不需要全部结果，可以使用 limit 参数限制返回结果的数量。其效果与 SQL 中的 limit 关键字类似，当搜索到的结果数量达到 limit 的限制时就停止搜索返回结果。

在该文档树中有 3 个 Tag 符合搜索条件，但结果只返回了两个，因为限制了返回数量：

```
>>> soup.find_all("a",limit = 2)
[<a class = "sister" href = "http://example.com/elsie" id = "link1"> Elsie</a>,
<a class = "sister" href = "http://example.com/lacie" id = "link2"> Lacie</a>]
```

在调用 Tag 的 find_all()方法时，BeautifulSoup 会检索当前 Tag 的所有子孙结点，如果只想搜索 Tag 的直接子结点，可以使用参数 recursive。

下面是一段简单的文档：

```
<html>
 <head>
  <title>
   The Dormouse's story
```

```
    </title>
  </head>
...
```

使用 recursive 参数的搜索结果如下：

```
>>> soup.html.find_all("title")
[<title>The Dormouse's story</title>]
>>> soup.html.find_all("title",recursive=False)
[]
```

<title>标签在<html>标签下，但并不是直接子结点，<head>标签才是直接子结点。在允许查询所有后代结点时 BeautifulSoup 能够查找到<title>标签，但是当使用了 recursive=False 之后只能查找直接子结点，这样就查不到<title>标签了。

下面来了解搜索方法 find()的细节，其格式如下：

```
find( name,attrs,recursive,string, ** kwargs)
```

find_all()方法将返回文档中符合条件的所有 Tag，尽管有时候用户只想得到一个结果。例如文档中只有一个<body>标签，那么使用 find_all()方法来查找<body>标签就不太合适，使用 find_all()方法并设置 limit=1 不如直接使用 find()方法。下面的代码是等价的：

**【例 6-32】** find()的介绍。

```
>>> soup.find_all('title',limit=1)
[<title>The Dormouse's story</title>]
>>> soup.find('title')
<title>The Dormouse's story</title>
```

唯一的区别是 find_all()方法的返回结果是只包含一个元素的列表，而 find()方法直接返回结果。

find_all()方法没有找到目标时返回空列表，find()方法没有找到目标时返回 None。例如：

```
>>> print(soup.find("nosuchtag"))
None
```

soup.head.title 是 Tag 的名字方法的简写。这个简写的原理就是多次调用当前 Tag 的 find()方法：

```
>>> soup.head.title
<title>The Dormouse's story</title>
>>> soup.find("head").find("title")
<title>The Dormouse's story</title>
```

## 本章小结

本章借助实例详细介绍了如何利用 urllib 和 BeautifulSoup4 模块来实现对网络数据的获取，主要包括以下几个方面：

(1) 构成网页的两种典型的组织方式——HTML 和 XML，其中 HTML 用于布局网页的架构，XML 用于传递或存储数据。

(2) 如何使用 urllib 的 urlopen()方法获取 URL 的网页内容，以及其他几种典型且实用的网页获取和存储方法。

(3) 如何利用 BeautifulSoup4 模块解析 HTML 和 XML 文档，重点介绍了 BeautifulSoup4 如何组织 HTML 和 XML 文档，以及如何使用 find_all()和 find()方法寻找数据。

## 习题

请利用 urllib 和 BeautifulSoup4 模块设计一个小型程序，实现对某个感兴趣的网页进行信息爬取的功能。

# 第 7 章

# 文件操作

**本章学习目标：**

- Python 打开文件的方法
- Python 打开文件的各种模式
- Python 读写文件的模式
- 用 Python 构建文本对话框

从系统磁盘中读取文件并将想要添加的内容写入已有文件是用户经常要做的操作。本章介绍 Python 读写文件的操作，包括读取文件、写文件、保存文件和使用文本对话框等。



视频讲解

## 7.1 文件的打开和关闭

### 7.1.1 打开文件

Python 提供了内置的 open()方法用于打开文件，用户可以使用 help()方法查看 open()的一些属性：

```
In [1]:
help(open)
Out[1]:
Help on built-in function open in module io:
open(file,mode='r',buffering=-1,encoding=None,errors=None,newline=None,closefd=
True,opener=None)
Open file and return a stream. Raise IOError upon failure.
```

下面对 open()的参数进行解释。

- file：文件所在的路径。
- mode：文件的打开模式。文件的打开模式有很多，如表 7.1 所示。

**表 7.1　打开文件的各种模式**

| 模式 | 描　述 |
| --- | --- |
| r | 打开一个文件为只读模式，文件指针位于该文件的开头。这是默认模式 |
| rb | 打开一个文件，只能以二进制格式读取，文件指针位于该文件的开头 |
| r+ | 打开用于读取和写入的文件，文件指针位于文件的开头 |
| rb+ | 打开用于读取和写入的二进制格式文件，文件指针在文件的开头 |
| w | 打开一个文件，只写。如果该文件存在，则覆盖该文件；如果该文件不存在，则在该路径下创建一个新文件，用于写入 |
| wb | 打开一个文件，只能以二进制格式写入。如果该文件存在，则覆盖该文件；如果该文件不存在，则在该路径下创建一个新文件，用于写入 |
| w+ | 打开用于写入和读取的文件。如果该文件存在，则覆盖该文件；如果该文件不存在，则在该路径下创建一个新文件，用于写入 |
| wb+ | 打开用于写入和读取的二进制格式文件。如果该文件存在，则覆盖该文件；如果该文件不存在，则在该路径下创建一个新文件，用于写入 |
| a | 打开文件，文件指针在该文件的末尾，也就是说该文件处于追加模式。如果该文件不存在，则在该路径下创建一个新文件，用于写入 |
| ab | 打开一个二进制格式文件，文件指针在该文件的末尾，也就是说该文件处于追加模式。如果该文件不存在，则在该路径下创建一个新文件，用于写入 |
| a+ | 打开一个追加和读取的文件，文件指针在该文件的末尾，该文件为追加模式。如果该文件不存在，则在该路径下创建一个新文件，用于读取和写入 |
| ab+ | 打开一个追加和读取的二进制文件，文件指针在该文件的末尾，该文件为追加模式。如果该文件不存在，则在该路径下创建一个新文件，用于读取和写入 |
| b | 以二进制的形式打开文件 |

- buffering：如果 buffering 的值被设置为 0，就不会有寄存；如果 buffering 的值取 1，在访问文件时会寄存；如果将 buffering 的值设置为大于 1 的整数，表明这就是寄存区的缓冲大小；如果取负值，寄存区的缓冲大小为系统默认。
- encoding：编码方式，默认为 None。

当文件被打开后会有一个 File 对象，用户可以通过该对象得到关于该文件的各种信息。例如：

```
file = open(file path, 'w + ')
```

表 7.2 列出了和 File 对象相关的所有属性：

**表 7.2　File 对象的属性**

| 属　　性 | 描　　述 |
|---|---|
| file. closed | 返回 True 表示文件已关闭，返回 False 表示文件未关闭 |
| file. mode | 返回被打开的文件的访问模式 |
| file. name | 返回文件的名称 |
| file. softspace | 如果用 print()输出后必须跟一个空格符，则返回 False，否则返回 True |

### 7.1.2　关闭文件

File 对象的 close()方法用于刷新缓冲区里任何还没有写入的信息，并关闭该文件，在这之后便不能再对文件进行写入操作了。当一个文件对象的引用被重新指定给另一个文件时，Python 会关闭之前的文件。用 close()方法关闭文件是一个很好的习惯。其语法格式如下：

```
file.close()
```

## 7.2　读写文件



视频讲解

### 7.2.1　从文件读取数据

File 对象提供了 3 个读文件的方法，即 read()、readline()和 readlines()。每种方法都可以接受一个变量，以限制每次读取的数据量，但它们通常不使用变量。read()每次读取整个文件，它通常用于将一个文件内容放入到一个字符串变量中。然而，当 read()读取的文件内容大于可用内存时，不可能接受这种处理。readline()和 readlines()之间的差别在于后者是一次性读取整个文件，和 read()一样，readlines()自动将文件内容解析成一个行的列表，该列表可以由 Python 的 for-in-结构进行处

理；另一方面，readline()每次只读取一行，通常比 readlines()慢很多。表 7.3 给出 File 对象的读取方法和描述。

表 7.3　读取文件的方法

| 方　法 | 描　述 |
|---|---|
| file.read([size]) | size 表示读取的长度，单位为字节。读取整个文件 |
| file.readline([size]) | 读取一行，每操作一次读取一行，读取长度为 size，若 size 的大小小于这一行的长度，则返回这一行的部分 |
| file.readlines([size]) | 把文件的每一行作为 list 的一个成员，读取后返回一个 list，读取的行数为 size，若 size 小于文件的总行数，则返回文件的部分行 |

当读取的文件很大时，经常使用 fileinput 模块：

```
import fileinput
for line in fileinput.input(file_path):
  print (line)
```

用户也可以直接使用 for 循环：

```
f = open(file_path)
for line in f.readlines():
  print (line)
```

用户还可以使用列表解析式：

```
[line for line in open(file_path).readlines()]
```

在使用 open()方法打开文件后一定要记得调用 close()方法关闭文件，可以用 try-finally 语句来确保最后能关闭文件，例如：

```
f = open(file_path)
try:
  for line in f.readlines():
    print (line)
finally:
  f.close()
```

**注意**：不能将 open()方法放在 try 里面，因为当打开文件出现异常时文件对象将无法指向 close()操作。

### 7.2.2 向文件写入数据

write()方法可以将任何字符串写入一个打开的文件中。需要注意的是，Python字符串可以是二进制数据，而不仅仅是文字，write()方法不会在字符串结尾添加换行符('\n')。

writelines()也可以将内容写入打开的文件中，但是和write()方法一样，writelines()也只是机械地写入，不会在每行后面添加任何东西。

## 7.3 文件对话框

### 7.3.1 基于win32ui构建文件对话框

从名字上看，win32ui模块是对Windows系统进行文件对话框操作的，该模块里面的CreateFileDialog()方法可以很方便、快捷地创建打开文件对话框。代码展示如下：

```
import win32ui

dlg = win32ui.CreateFileDialog(1)              #创建打开文件对话框
dlg.SetOFNInitialDir("D:\\python")             #设置打开文件对话框中的初始显示目录
dlg.DoModal()

filename = dlg.GetPathName()                   #获取选择的文件名称
print (filename)
```

其结果如图7.1所示。

CreateFileDialog()有几种内置方法，如表7.4所示。

**表7.4 CreateFileDialog()的几种内置方法**

| 方　　法 | 功　　能 |
| --- | --- |
| GetPathName() | 获取路径名称 |
| GetFileName() | 获取文件名称 |
| GetFileExt() | 获取文件扩展名 |
| GetFileTitle() | 获取文件标题 |
| GetPathNames() | 从文件对话框获取路径名称列表 |

续表

| 方　　法 | 功　　能 |
| --- | --- |
| GetReadOnlyPref() | 获取只读文件 |
| SetOFNTitle() | 设置对话框名 |
| SetOFNInitialDir() | 设置对话框的初始文件夹 |
| DoModal() | 为对话框创建一个模式窗口 |
| EndDialog() | 关闭一个模式对话框 |

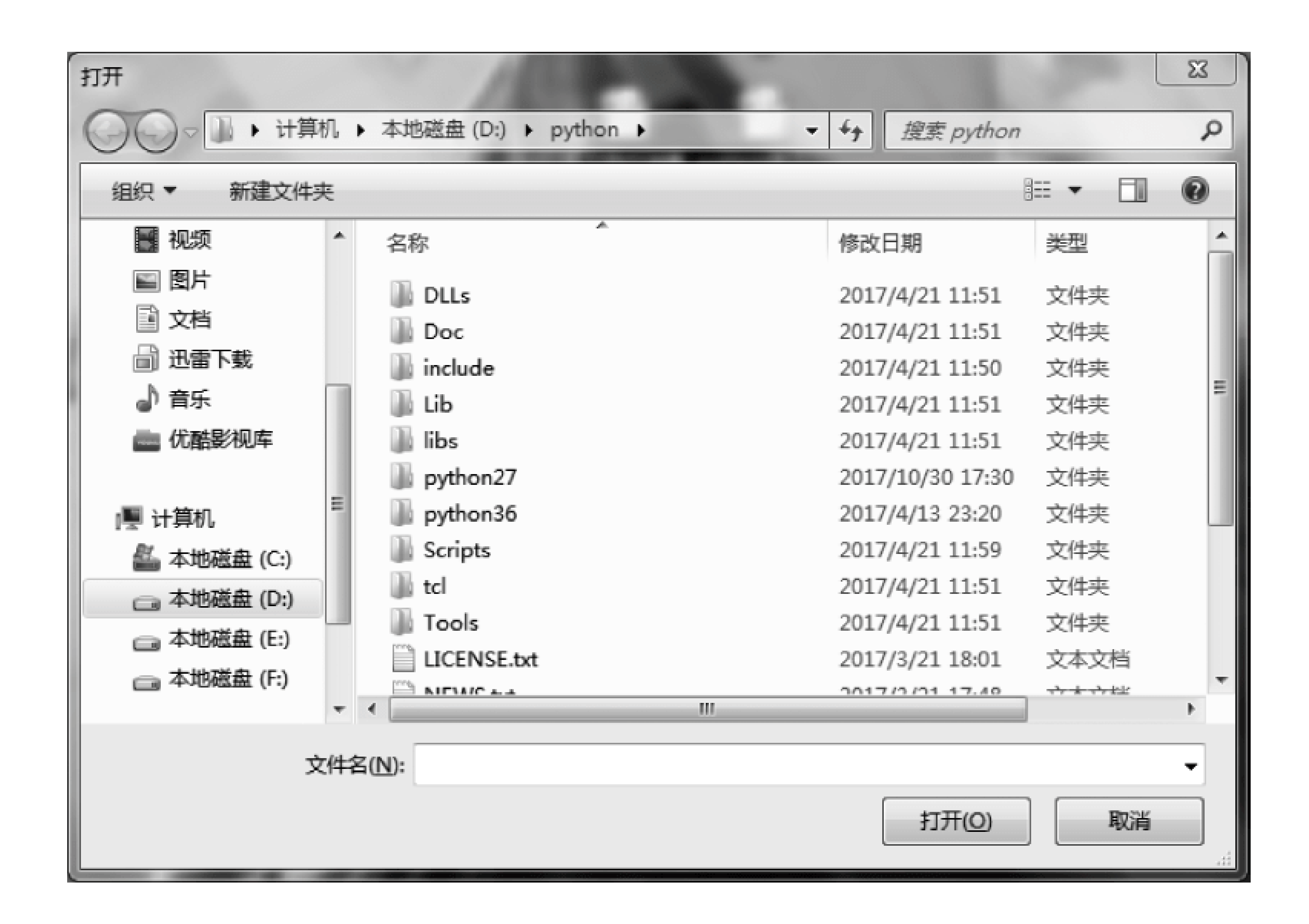



图 7.1　使用 win32ui 创建文件对话框

这个打开文件对话框的界面还是很友好的，这也是 Windows 风格的对话框，但是它的缺点也很明显，那就是只对 Windows 系统有效。当对其他系统进行打开或创建文件对话框的操作时，需要用到 tkFileDialog 模块。

## 7.3.2　基于 tkFileDialog 构建文件对话框

tkFileDialog 的功能和 win32ui 差不多，都是用于对文件对话框的操作，它的代码也很简单，代码展示如下：

```
import tkFileDialog
filename = tkFileDialog.askopenfilename(initialdir = 'E:/Python')
print (filename)
```

其得到的效果是和 win32ui 一样的。

表 7.5 列出了 tkFileDialog 的几种常用方法。

**表 7.5　tkFileDialog 的几种常用方法**

| 方　　法 | 功　　能 |
|---|---|
| askopenfile(mode='r', ** options) | 打开一个文本对话框,返回一个文本对象。若需要返回多个文本对象,使用 askopenfiles(mode='r', ** options),此时将以列表形式返回文件对象 |
| askopenfilename( ** options) | 获取文件路径和名称。若要获取多个文件路径和名称,使用 askopenfilenames( ** options),此时将以元组形式返回文件路径和名称 |
| asksaveasfile(mode='w', ** options) | 打开文本对话框,返回一个写文本对象 |
| asksaveasfilename() | 获取需保存文件的路径和名称 |
| askdirectory() | 选择一个文件夹 |

## 7.4　应用实例:文本文件的操作

**【例 7-1】** 使用 random 模块中的 randint()方法生成 1~122 的随机数,以产生字符对应的 ASCII 码,然后将满足大写字母、小写字母、数字和一些特殊符号(例如\n、\r、*、&、^、$)条件的字符逐一写进 test.txt 中,当光标到达 10001 时停止写入。

程序代码如下:

```
import random
with open('F:\\python book\\chapter7\\test.txt','w') as f:
  while 1:
    i = random.randint(1,122)
    x = chr(i)
    if x.isupper() or x.islower() \
      or x.isdigit() or x in ['\n','\r','*','&','^','$']:
        f.write(x)
    if f.tell() > 10000:
      break
```

运行该文件,会在 F 盘的 python book 文件夹下的 chapter7 目录中产生一个 test.txt 文件,在该文件中会写用户想要的内容,如图 7.2 和图 7.3 所示。

当然,还有许多创建文本文件的方法,读者也可以尝试自己编写。

以下实例均在上述代码产生的文本文件的基础上运行。




图 7.2　在 F 盘的 python book 文件夹下的 chapter7 目录中创建 test.txt 文件




图 7.3　test.txt 文件中的部分内容

**【例 7-2】** 逐个字节输出 test. txt 文件中的前 100 个字节字符和后 100 个字节字符。

程序代码如下：

```
with open('F:\\python book\\chapter7\\test.txt','r') as f:
  print(f.read(100))
  f.seek(9900) #将光标移动到倒数第 9900 的位置
  print (f.read())
```

**【例 7-3】** 逐行输出 test. txt 文件中的所有字符。

分析：这里能用很多方法实现，可以用 readlines()生成一个列表，或者直接迭代文本对象。下面给出两种实现方法。

程序代码 7-3-1(方法一)：

```
with open('F:\\python book\\chapter7\\test.txt','r') as f:
  lines = f.readlines()
  for line in lines:
    print (line)
```

程序代码 7-3-2(方法二)：

```
with open('F:\\python book\\chapter7\\test.txt','r') as f:
  for line in f:
    print (line)
```

方法一先产生一个由各行字符构成的列表，然后逐个打印出列表中的元素；方法二是利用了文本对象的迭代功能，直接对文本内容进行读取。相比较而言，由于方法一构建了一个列表，所以程序的运行更占内存，建议用户使用方法二，直接对文本对象进行迭代。

**【例 7-4】** 复制 test. txt 文件中的文本数据，生成一个新的文本文件。

分析：以读写模式打开需要复制的文件，将文本中的所有字符赋给一个新的变量，然后以写模式新建一个文本对象，将所有字符写入该文本中；或者逐字节或逐行将需要赋值的文本字符写入新文本。下面给出这两种实现方法。

程序代码 7-4-1(方法一)：

```
f = open('F:\\python book\\chapter7\\test.txt','r')
g = open('F:\\python book\\chapter7\\test1.txt','w')
a = f.read()
```

```
g.write(a)
f.close()
g.close()
```

程序代码 7-4-2(方法二)：

```
f = open('F:\\python book\\chapter7\\test.txt','r')
g = open('F:\\python book\\chapter7\\test1.txt','w')
for contents in f:
  g.write(contents)
f.close()
g.close()
```

**【例 7-5】** 统计 test.txt 文件中大写字母、小写字母和数字出现的频率。

分析：利用字符串的内置方法 isupper()、islower()和 isdigit()判断字符的类型，或者直接判断是否处于大写字母、小写字母和数字的范围。

程序代码 7-5-1(方法一)：

```
with open('F:\\python book\\chapter7\\test.txt','r') as f:
  u = 0
  l = 0
  d = 0
  for line in f.readlines():
    for content in line:
      if content.isupper():
        u += 1
      elif content.islower():
        l += 1
      elif content.isdigit():
        d += 1
  print ('大写字母有%d个，小写字母有%d个，数字有%d个' % (u,l,d))
```

程序代码 7-5-2(方法二)：

```
with open('F:\\python book\\chapter7\\test.txt','r') as f:
  u = 0
  l = 0
  d = 0
  for line in f.readlines():
    for content in line:
      if 'A'<= content <= 'Z':
        u += 1
      elif 'a'<= content <= 'z':
```

```
        l += 1
    elif '0'<= content <= '9':
        d += 1

print ('大写字母有 %d 个,小写字母有 %d 个, 数字有 %d 个' % (u,l,d))
```

**【例 7-6】** 将 test.txt 文件中的所有小写字母转换成大写字母，然后保存到 test_copy.txt 文件中。

分析：先以 w 模式创建一个空白的文本文件 test_copy.txt，然后将 test.txt 文件中的小写字母全部转换成大写字母，再写入 test_copy.txt 文件中。

程序代码：

```
f = open( 'F:\\python book\\chapter7\\test.txt', 'r')
g = open( 'F:\\python book\\chapter7\\test_copy.txt', 'w')
temp = ''                             #创建一个空字符串用于保存
for line in f.readlines():
  for content in line:
    if content.islower():
       content.upper()
       temp += content
    else:
       temp += content

g.write(temp)

f.close()
g.close()
```

## 本章小结

本章主要介绍了如何利用 Python 进行文本文件的操作，具体包括以下内容：

(1) 如何打开和关闭文本对象。

- 使用 open()函数可以创建新的文本文件或者打开已有的文本文件；
- 使用文本对象的 close()方法可以将缓存的文本数据存储到磁盘中，并关闭文本。

(2) 文本对象的几种模式。基本模式包括 r、w、a，分别对应读模式、写模式和附加模式。这 3 种模式可以和＋与 b 结合使用，从而实现附加的文本对象功能。

（3）文本对象常用的方法和属性。

（4）如何读取文本对象中的数据。

（5）如何往文本中写数据。

（6）构建文本对话框。

（7）几种典型的文本操作。

## 习题

1. 编写程序实现九九乘法表，并将其保存到一个 ex7_1.txt 文件中。

2. 编写程序，提示用户输入字符串，将所输入的字符串以及对应字符串的长度写入到 ex7_2.txt 中。

3. 在指定路径新建一个空白.txt 文件，并将“Hello Python”写到该文件中。

4. 读取习题 3 中的文件，并将“Welcome to”添加到“Hello”的后面。

# 第 8 章



# Python数据可视化

**本章学习目标：**

- 理解数据可视化的基本概念
- 熟练掌握使用 Matplotlib 绘制常用图表的方法
- 掌握 Matplotlib 中图表的定制方法

本章先向读者介绍数据可视化的基本概念和常用的数据图表，然后介绍 Python 的 Matplotlib 可视化库，重点介绍如何使用该库的 pyplot 对象制作点线图、柱状图等常用图表，接下来介绍图表定制的基本方法，最后讲述高级的图表定制。

## 8.1 数据可视化概念框架

### 8.1.1 数据可视化简介

受进化的影响和制约，人类的大脑在处理图像和处理数字的速度上有着天壤之别，这与计算机在处理二者的方式上截然不同。人类的眼睛是一对高带宽、巨量视觉信号输入的并行处理器，拥有超强的模式识别能力，配合超过 50% 的功能用于视觉感知相关处理的大脑，使得人类通过视觉获取数据比任何其他形式的获取方式更好。

例如，假如若干信息通过纯数字或文字向人类传达，可能需要数分钟或数小时才能传达完毕，但若是以形状、颜色、布局等图像形式进行传达，往往在数秒之内即可将信息传达完毕。数据可视化正是利用人类的这一天生技能来增强数据处理和组织的效率。

数据可视化是将数据以视觉形式表达出来的科学技术，通常指的是较为高级的技术方法。这些技术方法允许利用图形/图像处理、计算机视觉以及用户界面，通过表达、建模以及对立体、表面、属性和动画的显示，对数据加以可视化解释。借助于图形化手段，数据可视化技术能够清晰、有效地传达与沟通信息。

数据可视化与信息图形、信息可视化、科学可视化以及统计图形密切相关。当前，在研究、教学和开发领域，数据可视化是一个极为活跃而又关键的方面。“数据可视化”这条术语实现了成熟的科学可视化领域和较年轻的信息可视化领域的统一。

大多数人对统计数据了解甚少，基本统计方法(平均值、中位数、范围等)并不符合人类的认知天性。最著名的一个例子是 Anscombe 的四重奏。请试着观察图 8.1(a)中的 4 组数据，即使是受过专业统计学训练的人也很难从这些数据中看出各组的规律，可一旦将 4 组数据可视化出来(如图 8.1(b)所示)，规律就非常清楚了。

(a)

| Ⅰ | | Ⅱ | | Ⅲ | | Ⅳ | |
|---|---|---|---|---|---|---|---|
| *x* | *y* | *x* | *y* | *x* | *y* | *x* | *y* |
| 10 | 8.04 | 10 | 9.14 | 10 | 7.46 | 8 | 6.58 |
| 8 | 6.95 | 8 | 8.14 | 8 | 6.77 | 8 | 5.76 |
| 13 | 7.58 | 13 | 8.74 | 13 | 12.74 | 8 | 7.71 |
| 9 | 8.81 | 9 | 8.77 | 9 | 7.11 | 8 | 8.84 |
| 11 | 8.33 | 11 | 9.26 | 11 | 7.81 | 8 | 8.47 |
| 14 | 9.96 | 14 | 8.10 | 14 | 8.84 | 8 | 7.04 |
| 6 | 7.24 | 6 | 6.13 | 6 | 6.08 | 8 | 5.25 |
| 4 | 4.26 | 4 | 3.10 | 4 | 5.39 | 19 | 12.5 |
| 12 | 10.84 | 12 | 9.13 | 12 | 8.15 | 8 | 5.56 |
| 7 | 4.82 | 7 | 7.26 | 7 | 6.42 | 8 | 7.91 |
| 5 | 5.68 | 5 | 4.74 | 5 | 5.73 | 8 | 6.89 |

(b)

图 8.1　Anscombe 四重奏可视化图表示例

人们在日常生活中所说的数据可视化大多指狭义的数据可视化以及部分信息可视化。根据数据类型和性质的差异，数据可视化经常分为以下几种类型。

(1) 统计数据可视化：用于对统计数据(例如平均值、中位数等)进行展示和分析，或对原始数据进行可视化统计，用于可视化的图表包括柱状图、饼图、直方图、散点图、折线图等。目前常见的可视化工具都具有统计数据可视化功能，例如 Excel、

ECharts、D3.js 等，都可用于展示、分析统计数据。

（2）关系数据可视化：主要表现为结点和边的关系，例如流程图、网络图、UML 图、力导图等。常见的关系可视化工具有 Visio、JointJS、GoJS 等。

（3）地理空间数据可视化：地理空间通常特指真实的人类生活空间，地理空间数据描述了一个对象在空间中的位置。在移动互联网时代，移动设备和传感器的广泛使用使得每时每刻都产生着海量的地理空间数据。地理空间可视化通常以将地理要素显示在地图上的方式进行表现，常见工具有 ArcGIS、Leaflet、百度/高德地图 API 等。

### 8.1.2 数据可视化常用图表

常用的数据可视化方法很多，包括传统的数据分析图表（例如柱状图、饼图）和现代化视觉图表（例如信息图、交互式地图等），其中有一些共同的视觉要素。

- 坐标：数值的位置被对应到直角坐标系或极坐标系上。
- 大小：数据的大小被对应到图形的大小。
- 色彩：数值的分类和界限等被对应到不同的颜色。
- 标签：数值的特征用标签来标记。
- 关联：数值之间的联系用关联线条等连接起来。

目前，基于统计的数据可视化常用的图表主要有柱状图（条形图）、折线图、饼图以及它们的变形种类，此外还有很多用于特定用途的专用图表。

#### 1. 柱状图

柱状图是数据可视化中最常使用的图表，可用于单个或多个维度的比较。文本维度/时间维度通常作为 X 轴，数值型维度通常作为 Y 轴。柱子的长短可直观反映出数值的大小，以便于迅速的比较。此外，若有第 3 个维度，可将柱子分组并通过柱子的颜色加以区分。

1）基础柱状图

柱状图可以表现每条数据的具体值，容易比较大小，每一组数据建议只展示 1～3 列数据，太多会导致对比困难，图形混乱。柱状图可以是水平排列的，也可以是垂直排列的，水平的柱状图有时候也称为条形图，如图 8.2 所示。

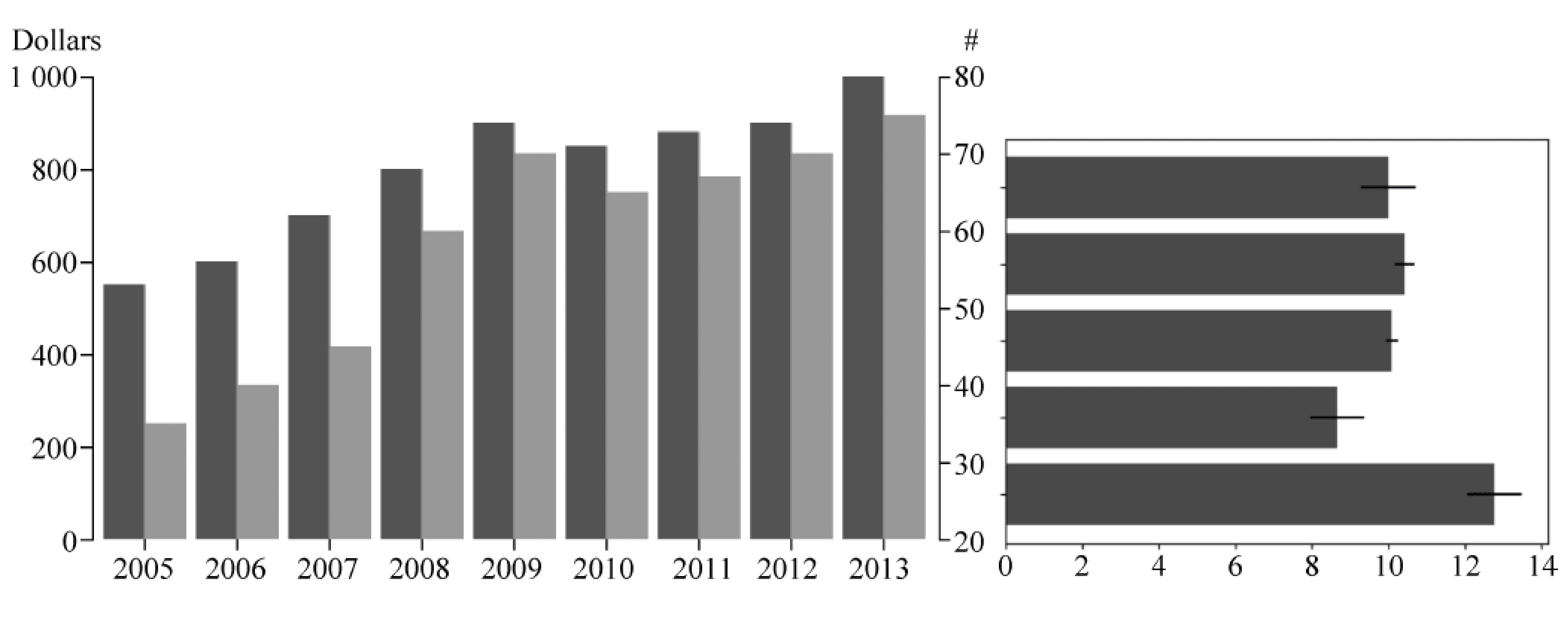



图 8.2　柱状图(条形图)示例

2）堆积柱状图(堆积条形图)

这是柱状图的一个变形类型,用于比较组内的占比情况,同时可以比较各组每部分及整体大小,如图 8.3 所示。

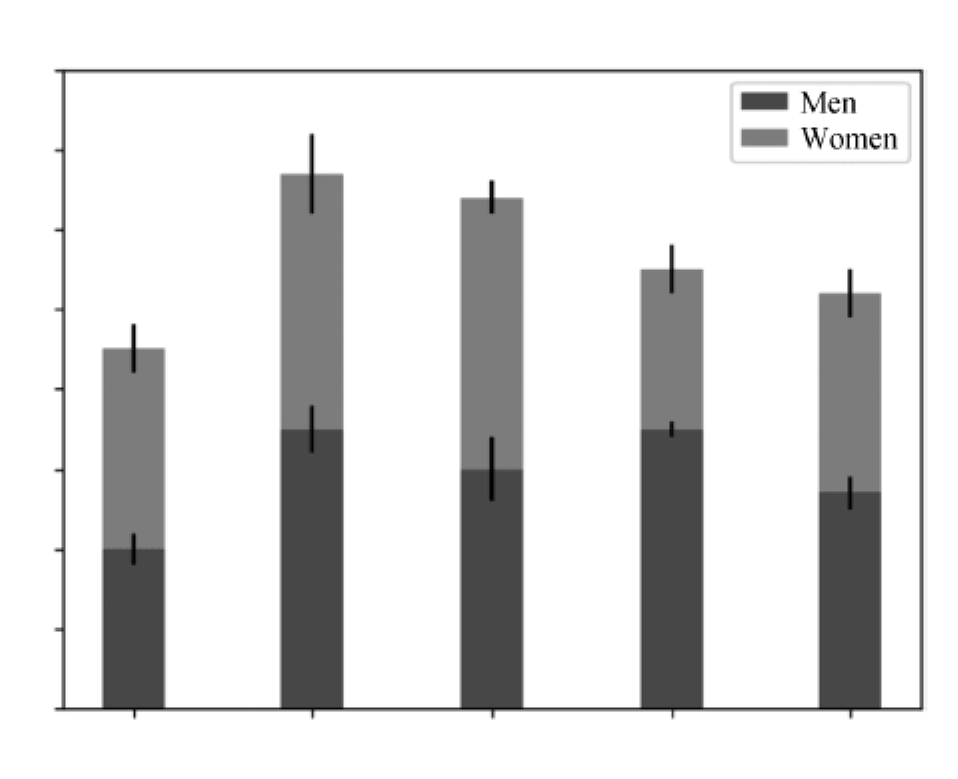



图 8.3　堆积柱状图示例

3）含正负值的条形图

以 X 轴 0 坐标为基准,向左右两边伸出的条状,可表达出具有正负值的数据,如图 8.4 所示。

**2. 折线图**

折线图最常用于观察数据的趋势,因此它和时间密不可分。当用户想要了解某一维度在时间上的规律或者趋势时就可以使用折线图,例如气温记录图、心电图都是常见的折线图,如图 8.5 所示。

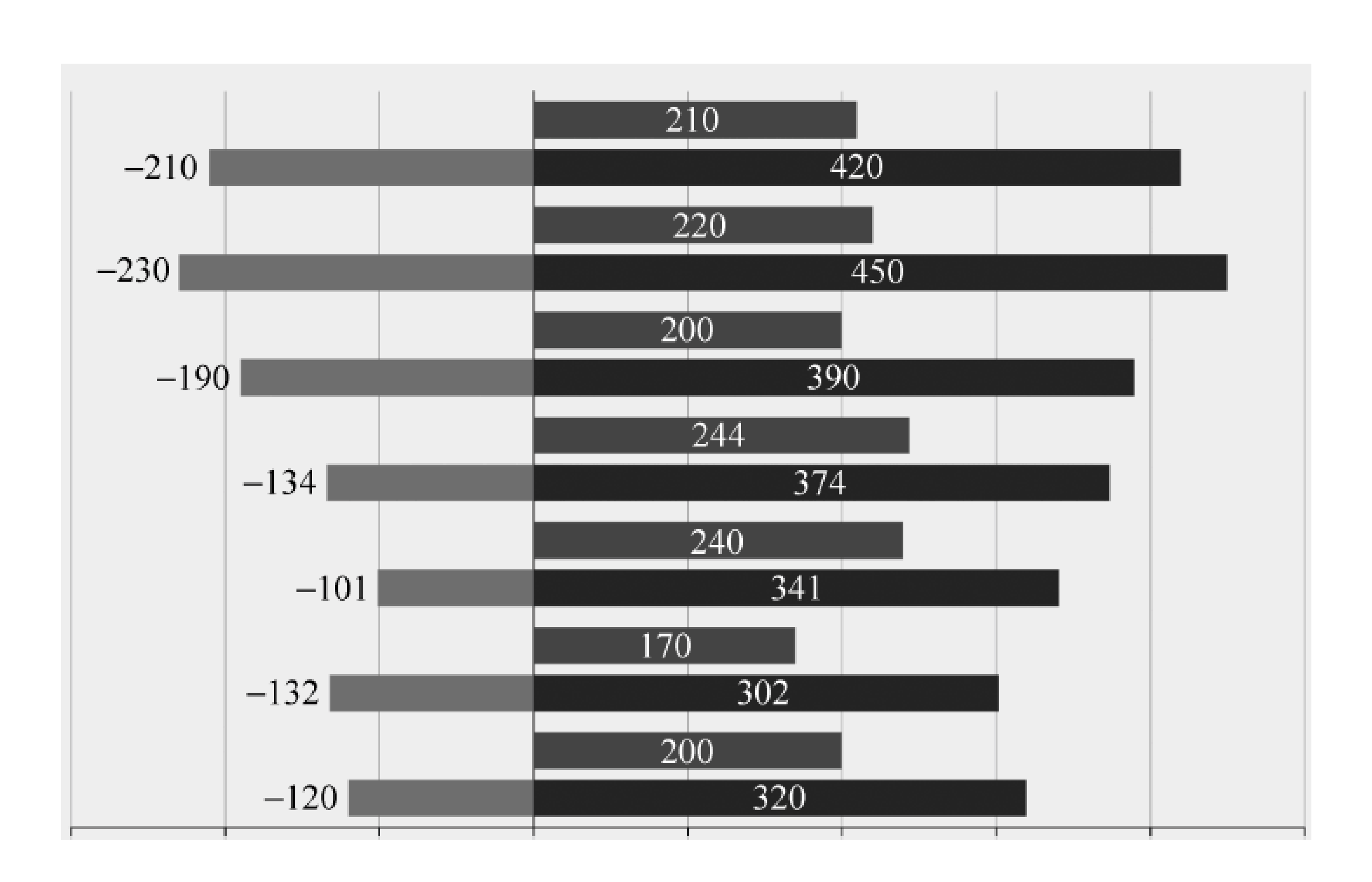



图 8.4 含正负值的条形图示例

**3. 饼图(环图)**

饼图在一个圆形上展示多组数据,每组数据的数值表达为其所在圆弧的相对角度(如图 8.6 所示),从而表现各组数据的占比情况。需要注意的是,饼图的使用限制比较大,它擅长表达某一占比较大的类别,但是不擅长对比,例如 30%和 35%在饼图上难以凭人的肉眼区分。此外,当类别过多时,也不适合在饼图上表达。

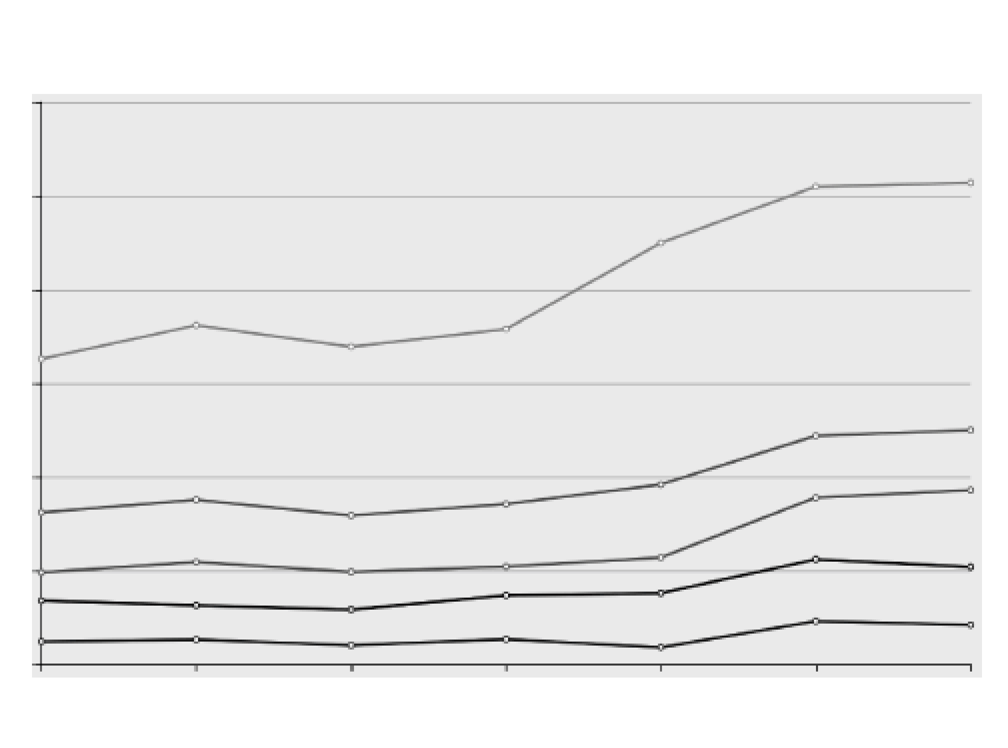


图 8.5 折线图示例

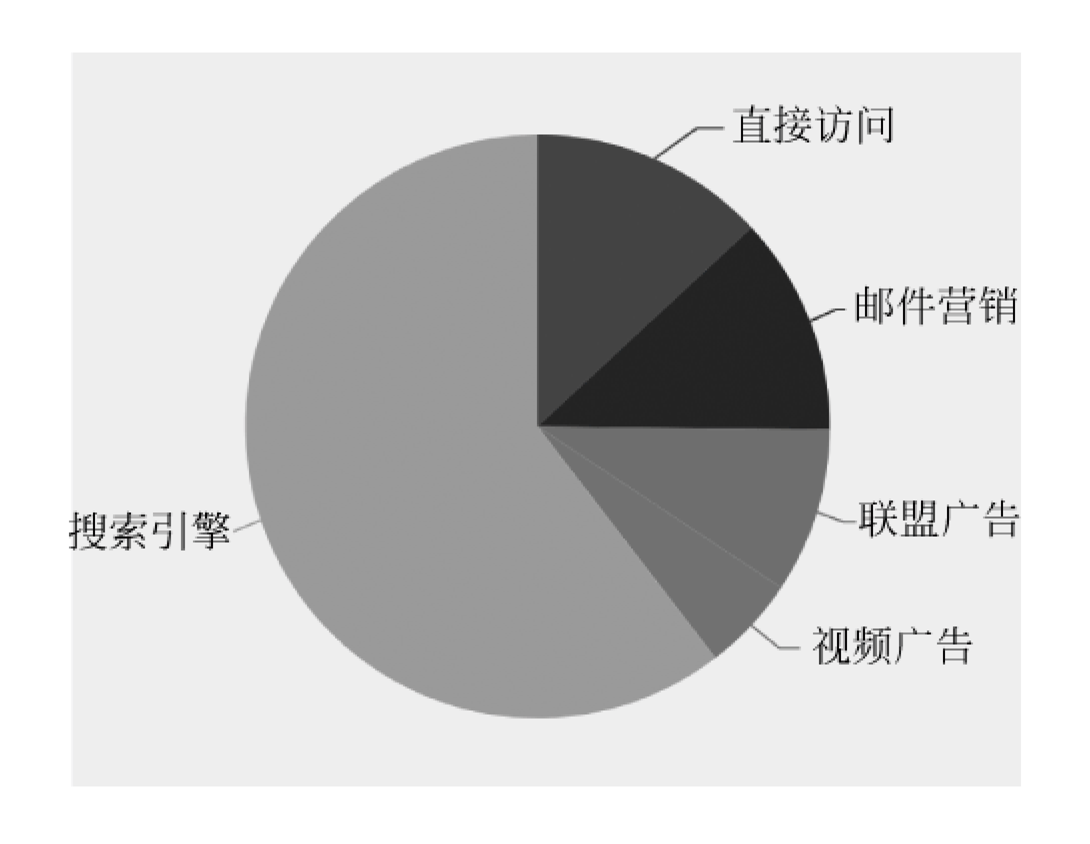



图 8.6 饼图示例

**4. 散点图**

散点图在日常报表中用得较少,但是在数据分析中极为常见,如图 8.7 所示。散点图通过坐标轴表示两个变量之间的关系。绘制它依赖大量数据点的分布。尤其是

对于大数据量，散点图会有更直观和精准的结果，例如统计中的回归分析或数据挖掘中的聚类。散点图的优势在于易于揭示数据间的关系，发现变量与变量之间的关联。

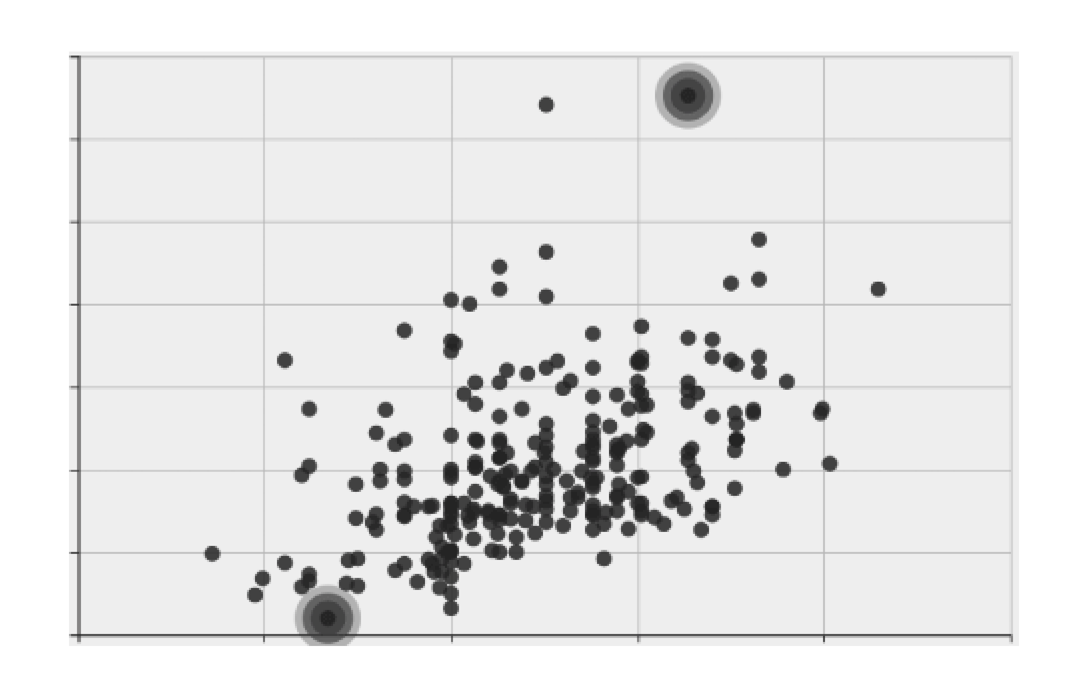


图 8.7　散点图示例

#### 5. 气泡图

气泡图与散点图类似，并在散点图的基础上为各个点赋予面积，使用点的面积大小来表达第 3 个维度的信息，如图 8.8 所示。

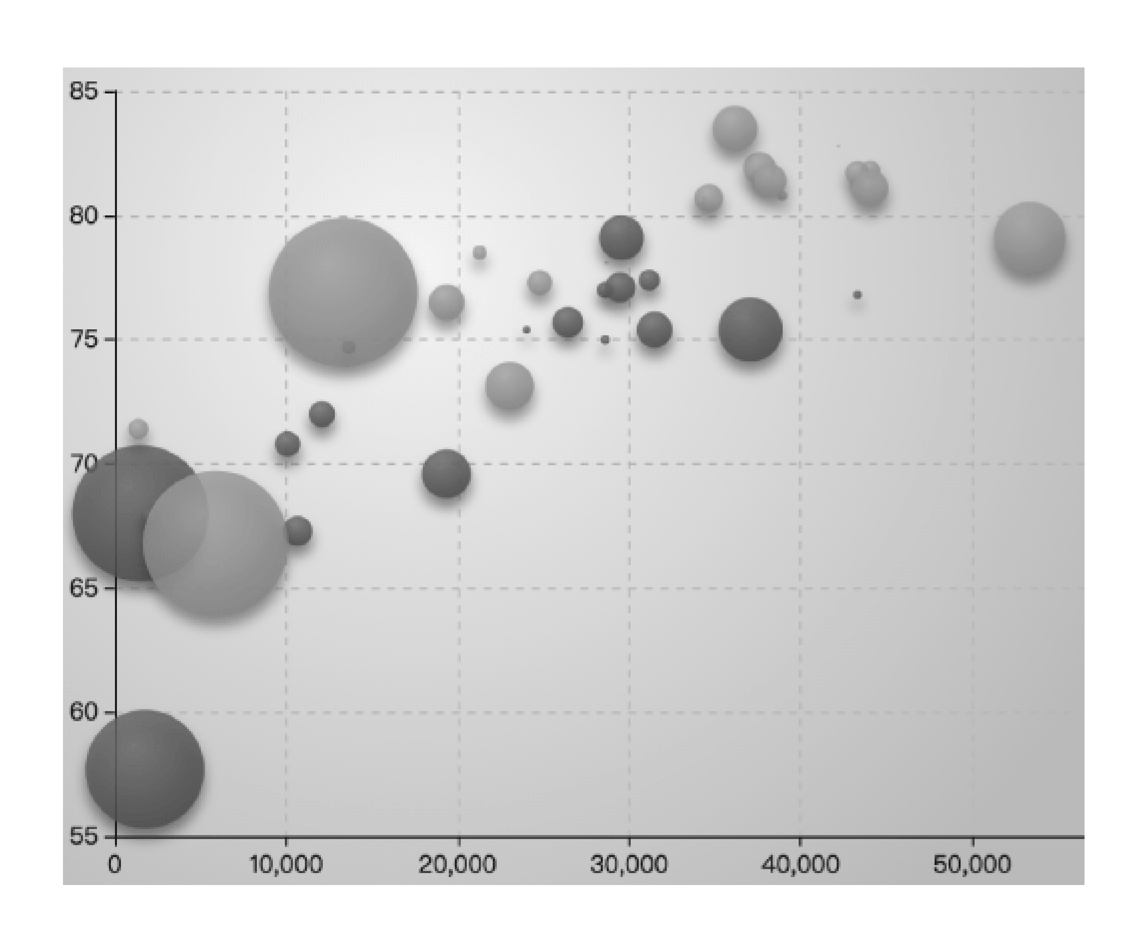



图 8.8　气泡图示例

### 8.1.3　Python 数据可视化环境准备

得益于 Python 强大的可扩展性，在 Python 中有很多强大的图形库供用户选择。

#### 1. Matplotlib

Matplotlib 是 Python 中最流行的 2D 绘图库，它可以在各种平台上以各种硬拷

贝格式交互式地生成具有出版品质的图形。Matplotlib 参考了 MATLAB 的界面和函数，提供与其相似的绘图方式。对于高级用户，可以通过面向对象的界面或 MATLAB 用户熟悉的一组函数完全控制线条样式、字体属性、轴属性等，只需几行代码即可使用 Matplotlib 将数据生成柱状图、饼图、散点图等常用图表。

**2. Seaborn**

Seaborn 是一个基于 Matplotlib 的高级可视化效果库，偏向于统计作图。因此，其针对的主要是数据挖掘和机器学习中的变量特征选取。相比 Matplotlib，Seaborn 的语法相对简单，绘制出来的图无须花很多时间去修饰。相对地，它的绘图方式比较局限，不够灵活。

**3. Bokeh**

Bokeh 是基于 JavaScript 来实现交互的可视化库，它可以在 Web 浏览器中实现美观的视觉效果。但是其缺点也较为明显：一是版本时常更新，有时语法不向下兼容；二是语法晦涩，相比 Matplotlib 而言更加复杂和难以理解。

**4. ggplot**

ggplot 是基于 R 语言中的 ggplot2 绘图库所制作的 Python 版本。ggplot2 是 R 语言中大名鼎鼎的绘图库，功能强大且应用广泛。如果用户对 R 语言和 ggplot2 比较熟悉，在 Python 中使用 ggplot 会更加得心应手。

**5. Plotly**

Plotly 也是一个做可视化交互的库。它不仅支持 Python，还支持 R 语言。Plotly 的优点是能提供 Web 在线交互，配色也更为美观。

总之，Python 中的绘图库众多，各有特点，但是 Matplotlib 是最基础、最常用、最强大的 Python 可视化库。如果要学习 Python 数据可视化，那么 Matplotlib 应该说是不二之选。本书将以 Matplotlib 作为学习 Python 可视化的主要工具。

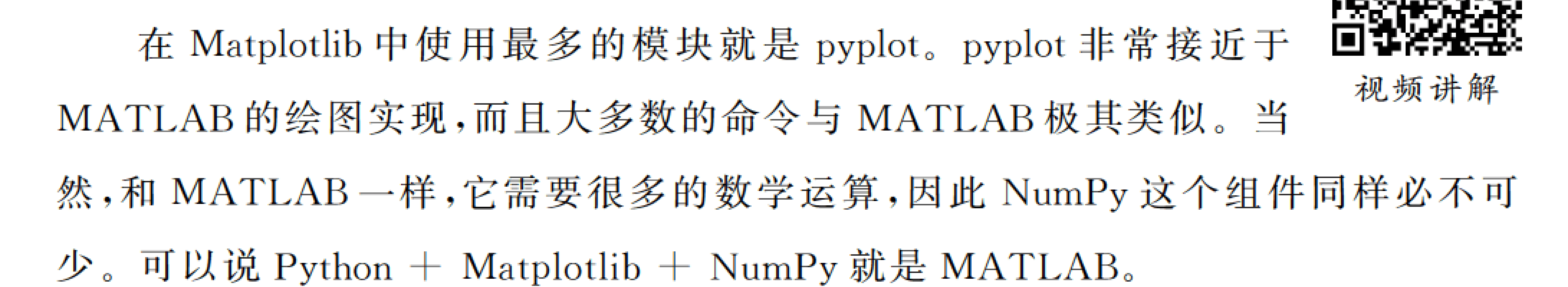
在操作系统中进入命令行工具，输入“pip3 install matplotlib”，或者访问 Matplotlib 的官方网站“https://pypi.python.org/pypi/matplotlib”，查找并下载与 Python 版本相配的 wheel 文件，安装之后测试 Matplotlib 是否安装成功：

```
import matplotlib
```

导入 Matplotlib，如果没有出现任何错误消息，就说明该系统安装了 Matplotlib。

## 8.2　绘制图表

### 8.2.1　Matplotlib API 入门



视频讲解

在 Matplotlib 中使用最多的模块就是 pyplot。pyplot 非常接近于 MATLAB 的绘图实现，而且大多数的命令与 MATLAB 极其类似。当然，和 MATLAB 一样，它需要很多的数学运算，因此 NumPy 这个组件同样必不可少。可以说 Python ＋ Matplotlib ＋ NumPy 就是 MATLAB。

首先，由于 Matplotlib 名称较长，所以建议在引入包时使用别名，这样方便以后对模块的使用，一般以以下两句开始：

```
import numpy as np
import matplotlib.pyplot as plt
```

这里导入了 Numpy 和 Matplotlib 的 pyplot 两个模块，并分别使用 np 和 plt 作为二者的别名。

下列代码使用 Matplotlib 绘制一个正弦和一个余弦函数曲线，这里暂不对图表做任何定制，使用默认的绘图属性绘图。

```
#创建 X 轴的数据：从 -π 到 π 的 256 个等差数字
x = np.linspace(-np.pi,np.pi,256,endpoint = True)

#使用 cos 和 sin 函数以 x 为自变量创建 C 和 S
C, S = np.cos(x),np.sin(x)

#使用 plot()分别绘制正弦和余弦函数
plt.plot(x,C)
plt.plot(x,S)
plt.show()
```

显示的结果如图 8.9 所示。

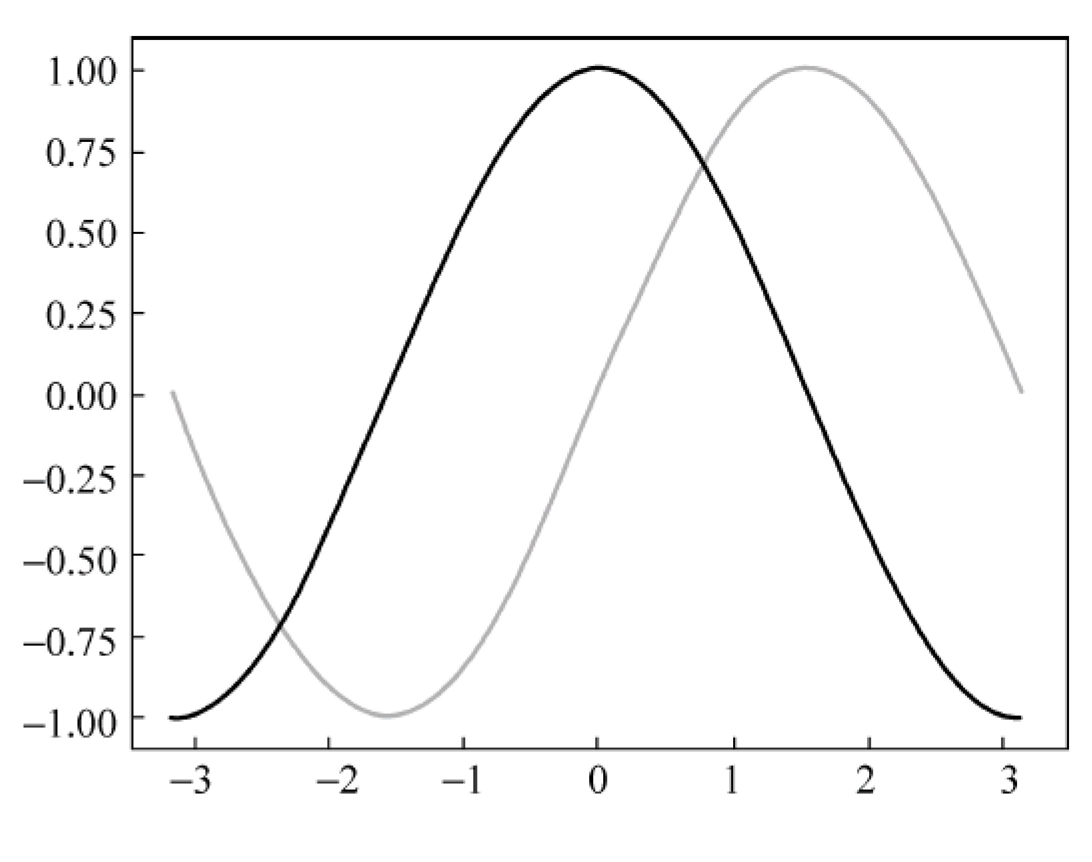



图 8.9　正弦与余弦曲线

在 Matplotlib 中，一个基本图表通常包括以下元素。

(1) Figure：Figure 指的是图像窗口，它是包裹 Axes、Title、Legend 等组件的最外层窗口。Figure 中最主要的元素是 Axes(子图)。在一个 Figure 中可以有多个子图，但至少要有一个能够显示内容的子图。

(2) Axes：Axes 指轴域/子图，是带有数据的图像区域，位于 Figure 里面。用户可以将 Axes 理解为覆盖在 Figure 上的小面板，用于显示实际的图表。

(3) Axis：轴，分为 X 轴和 Y 轴。轴上包含刻度和刻度标签。

此外，一个完整的图表通常还包含 Title(图表标题)和 Legend(图例)。

## 8.2.2　创建图表

视频讲解

### 1. 折线图

plot()是 pyplot 中最常用的函数，可用于创建一张点线图表，并通过参数对点标记和线条的样式进行设置。该函数的声明如下：

```
matplotlib.pyplot.plot( * args,  ** kwargs)
```

args 参数的长度是不定的，可以设置多个属性，包含多个 x、y；也可以设置折线的对应属性，例如颜色、线宽等。

plot(x1,y1,x2,y2,x3,y3,…)表示在同一幅图中显示多条折线，x1、y1 等均为数

值列表，代表坐标。

kwargs 参数主要用于设置图形要素的属性。例如：

```
plot(x,y,label = "red",color = "r",linestyle = " - ",linewidth = 5,marker = "^",markersize = 20)
```

这里分别设置了标签颜色、折线颜色、折线线型和线宽、标记点符号、标记点大小。

对于标注和线条的样式，可以通过简单的字符来表示，如表 8.1 所示。

**表 8.1 标注和线条样式的字符表示**

| 字 符 | 描 述 | 字 符 | 描 述 |
|---|---|---|---|
| '—' | 实线 | 's' | 方块符号 |
| '——' | 虚线 | 'p' | 五角形符号 |
| '—.' | 点横虚线 | '*' | 星星符号 |
| ':' | 点线 | 'h' | 六边形符号 |
| '.' | 点符号 | 'H' | 六边形符号 |
| ',' | 像素符号 | '+' | 加号 |
| 'o' | 圆圈符号 | 'x' | X 符号 |
| 'v' | 下三角符号 | 'D' | 菱形符号 |
| '^' | 上三角符号 | 'd' | 细菱形符号 |
| '<' | 左三角符号 | '\|' | 竖线符号 |
| '>' | 右三角符号 | '_' | 横线符号 |

支持的标注/线条颜色如表 8.2 所示。

**表 8.2 支持的标注/线条颜色**

| 符 号 | 颜 色 | 符 号 | 颜 色 |
|---|---|---|---|
| 'b' | blue | 'm' | magenta |
| 'g' | green | 'y' | yellow |
| 'r' | red | 'k' | black |
| 'c' | cyan | 'w' | white |

除了符号，线条的颜色可以通过其他方式设置，例如十六进制字符串('#FFFFFF')。

下列代码绘制了一段最基本的折线图：

```
import numpy as np
import matplotlib.pyplot as plt
```

```
x = np.linspace( - 2, 6, 5)
y1 = x + 3                          # 一条直线
y2 = 3 - x                          # 另一条直线
plt.figure()                        # 定义一个图像窗口
plt.plot(x, y1)                     # 绘制直线 1
plt.plot(x, y2)                     # 绘制直线 2
plt.show()
```

np.linspace()函数用于创建一个 numpy 数组,该数组包含−2～6 的等差的 5 个值,即−2、0、2、4、6。事实上,由于绘制的是直线,即使只有两个值(例如−2 和 6),也不影响最终的显示效果。但是如果绘制的是曲线,那么点越多,密度越大,则最终绘制出的曲线就越平滑。y1 和 y2 分别是这 5 个值对应直线的函数值组成的 numpy 数组。plt.figure()函数用于定义一个图像窗口,plot()函数用于绘图。最后通过调用 show()函数将图形呈现出来,如图 8.10 所示。

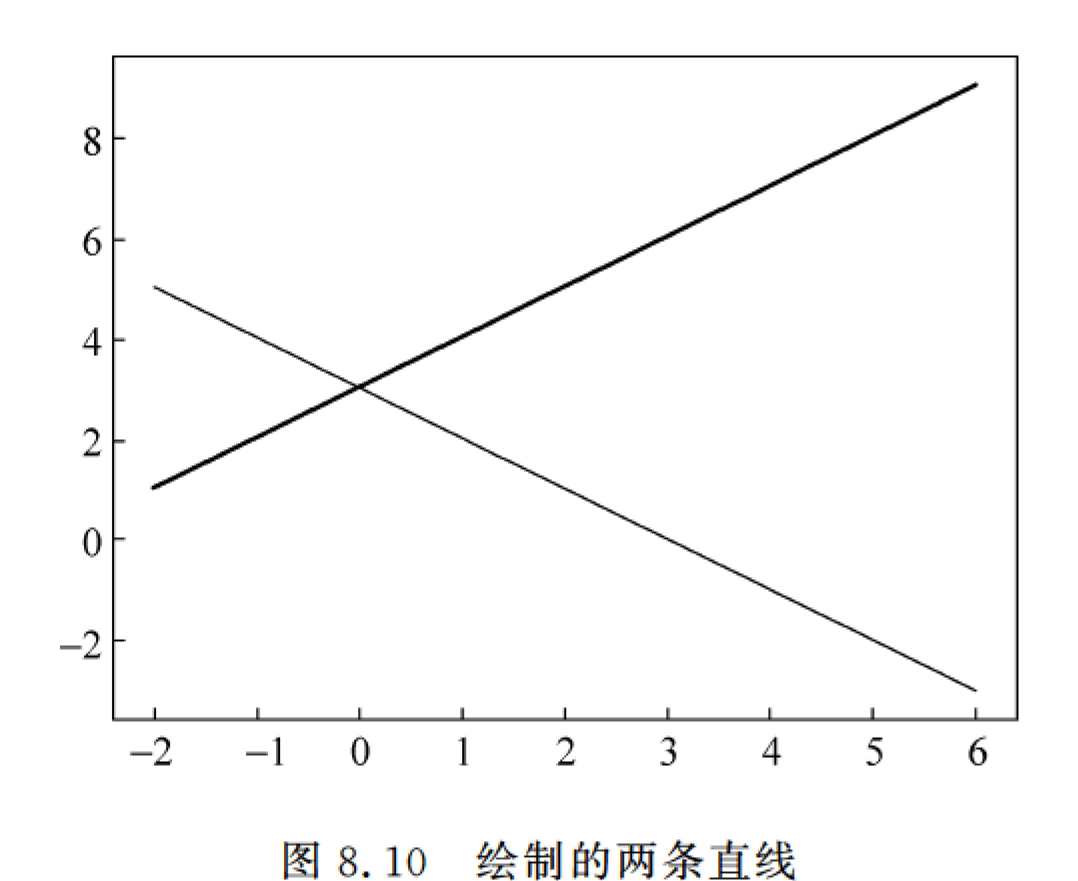



图 8.10 绘制的两条直线

上述代码中没有为 plot()函数提供任何样式参数,因此 Matplotlib 以默认样式将图形显示出来。为了美观,可以给两条线设置期望的颜色和线条类型,同时还给纵轴和横轴设置了上下限。

```
import numpy as np
import matplotlib.pyplot as plt

# 创建一个点数为 8×6 的窗口, 并设置分辨率为 80 像素/英寸
plt.figure(figsize = (8, 6), dpi = 80)
```

```
x = np.linspace( - 2, 6, 5)
y1 = x + 3                    # 直线 1
y2 = 3 - x                    # 直线 2

# 绘制蓝色、宽度为 1 个像素的实线
plt.plot(x, y1, color = "blue", linewidth = 1.0, linestyle = " - ")
# 绘制紫色、宽度为 2 个像素的虚线
plt.plot(x, y2, color = " # 800080", linewidth = 2.0, linestyle = " -- ")

# 设置横轴的上下限为 - 1～6
plt.xlim( - 1, 6)
# 设置纵轴的上下限为 - 2～10
plt.ylim( - 2, 10)

plt.show()
```

绘制后的图像如图 8.11 所示。

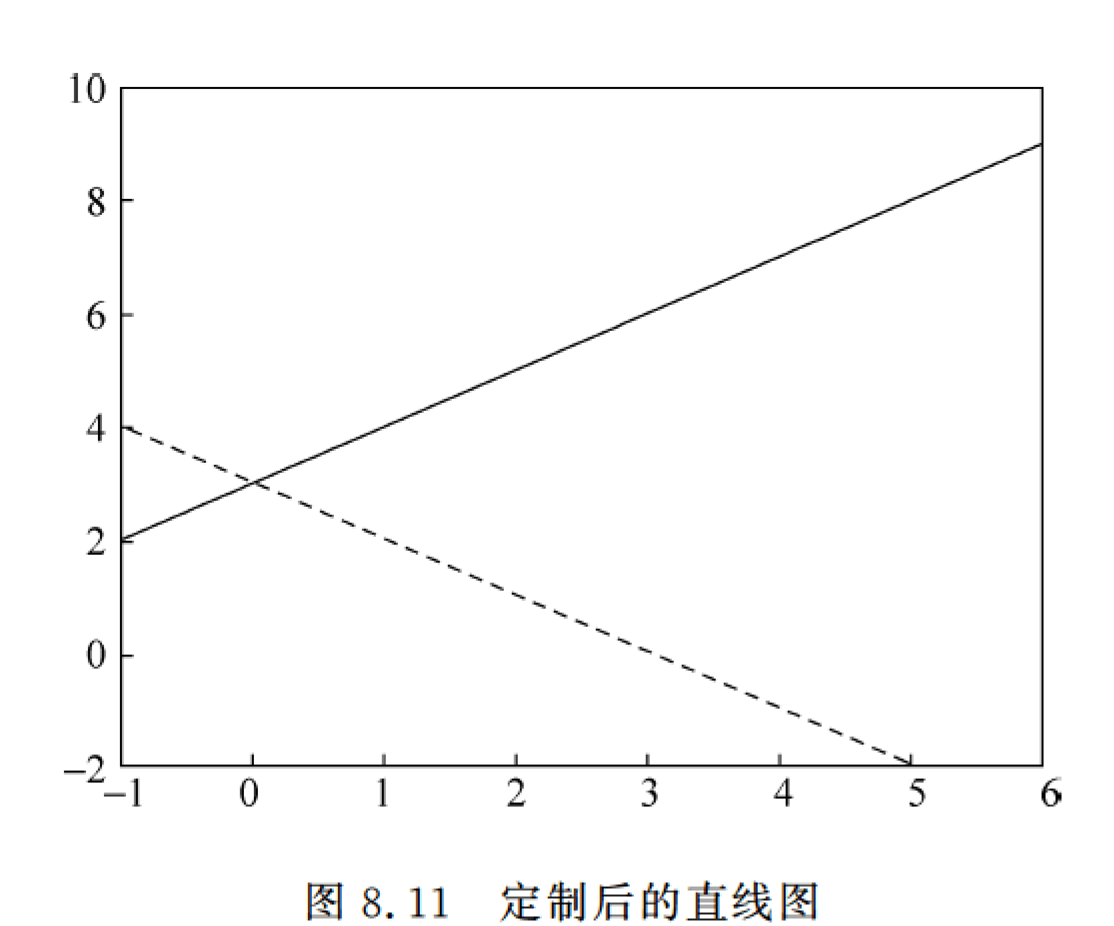



图 8.11 定制后的直线图

**2. 柱状图**

通过 pyplot 的 bar()函数可以生成柱状图。bar()的构造函数如下：

```
bar(x,height,width, * ,align = 'center', ** kwargs)
```

其中，x 是包含所有柱子的下标的列表。height 是包含所有柱子的高度值的列表。width 表示每个柱子的宽度，如果指定一个固定值，那么所有的柱子都是相同宽

度；如果设置一个列表，那么可以分别对每个柱子设定不同的宽度。align 表示柱子的对齐方式，有两个可选值——center 和 edge。center 表示每根柱子是根据下标来对齐；edge 则表示每根柱子全部以下标为起点，显示到下标的右边。如果不指定 align 参数，默认值是 center。

下列代码演示了如何使用 bar()函数绘制一个柱状图：

```
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize = (8, 6), dpi = 80)

# 柱子总数
N = 6
# 包含每个柱子对应值的序列
values = (5, 16, 20, 25, 23, 28)

# 包含每个柱子下标的序列
index = np.arange(N)

# 柱子的宽度
width = 0.35

# 绘制柱状图，每根柱子的颜色为蓝色
p2 = plt.bar(index, values, width, label = "月均气温", color = "#87CEFA")

# 设置横轴标签
plt.xlabel('月份')
# 设置纵轴标签
plt.ylabel('温度（摄氏度）')

# 添加标题
plt.title('月均气温')

# 添加纵横轴的刻度
plt.xticks(index, ('一月', '二月', '三月', '四月', '五月', '六月'))
plt.yticks(np.arange(0, 50, 10))

# 添加图例
plt.legend(loc = "upper right")
```

绘制后的图像如图 8.12 所示。

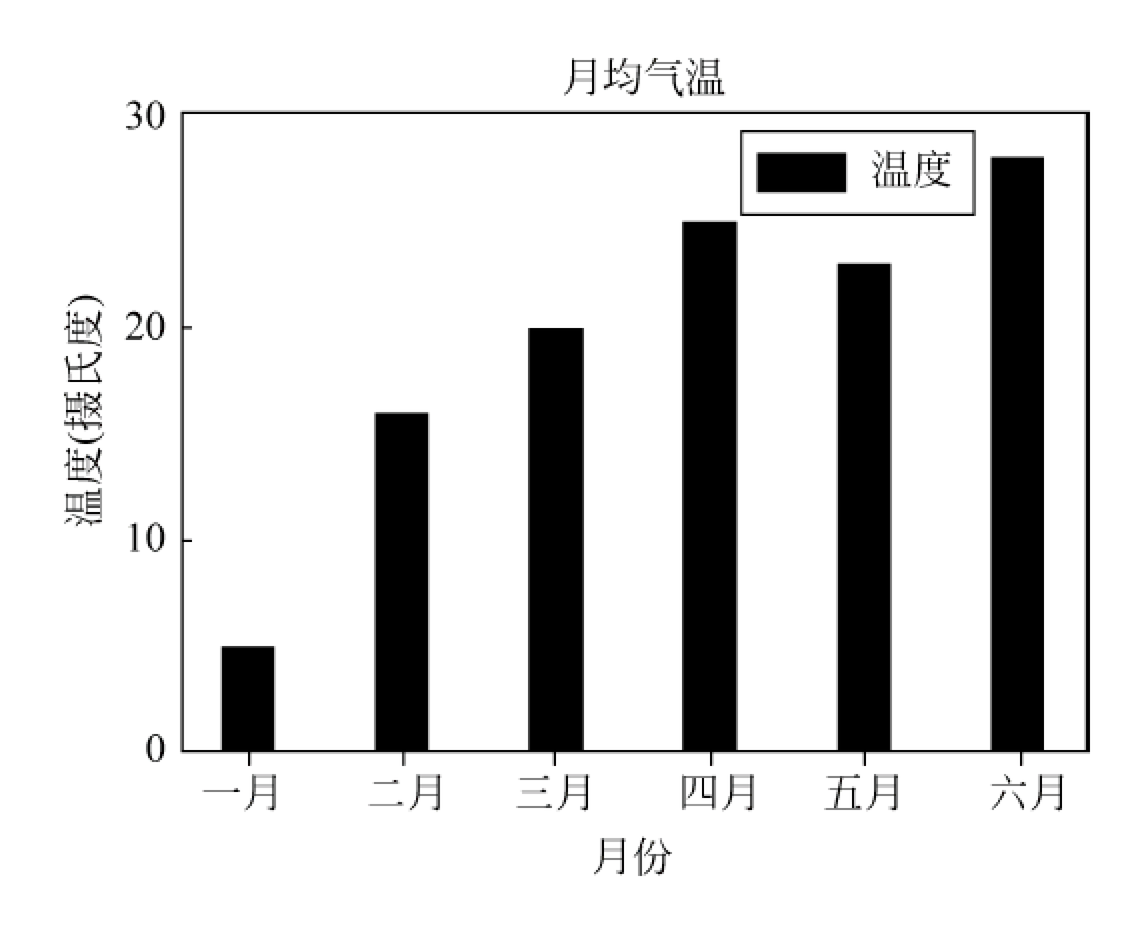



图 8.12　柱状图示例

### 3. 饼图

通过 pyplot 的 pie()函数可以生成饼图。pie()的构造函数如下：

```
matplotlib.pyplot.pie(x, explode = None, labels = None, colors = None, autopct = None,
pctdistance = 0.6, shadow = False, labeldistance = 1.1, startangle = None, radius = None,
counterclock = True, wedgeprops = None, textprops = None, center = (0, 0), frame = False,
rotatelabels = False, *, data = None)
```

该构造函数参数较多，其中，x 为数值列表，作为饼图的数据；explode 指定饼图中突出的分片；labels 设置各个分片的标签；colors 设置各个分片的颜色；autopct 设置标签中的数字格式；shadow 设置是否有阴影；startangle 设置从哪个角度开始绘制圆饼。

下列代码演示了一个最基本的饼图：

```
import matplotlib.pyplot as plt

labels = '大一', '大二', '大三', '大四'
sizes = [15, 30, 45, 10]
explode = (0, 0.1, 0, 0)            #将"大二"突出显示

fig1, ax1 = plt.subplots()
ax1.pie(sizes, explode = explode, labels = labels, autopct = '%1.1f%%',
        shadow = True, startangle = 90)
ax1.axis('equal')                   #确保饼图是个圆形

plt.show()
```

绘制后的图像如图 8.13 所示。

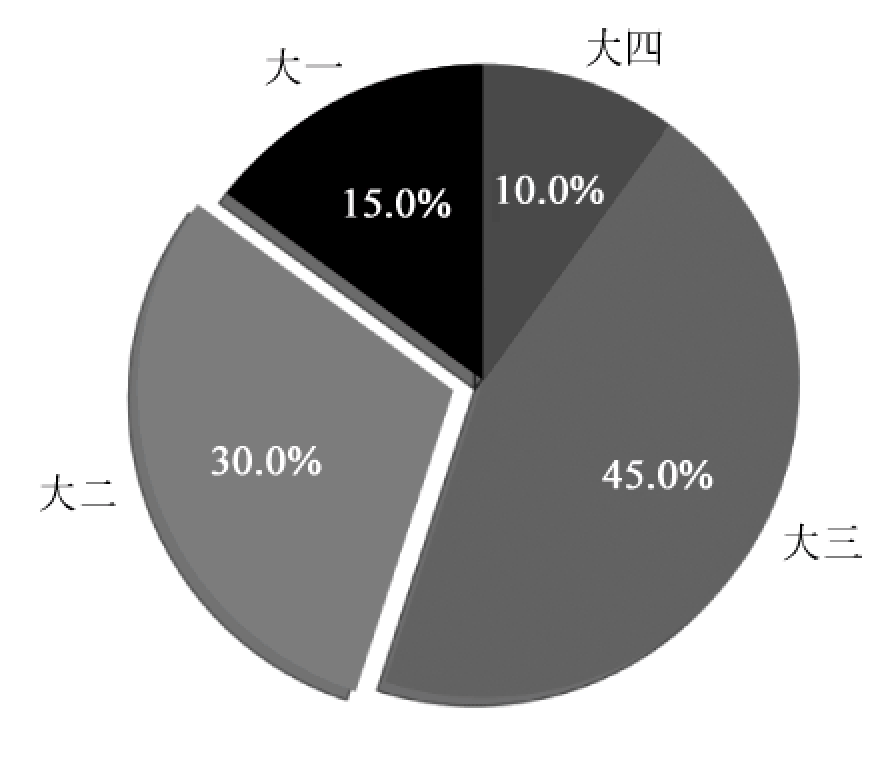



图 8.13 饼图示例

## 8.2.3 图表定制

视频讲解

### 1. 设置横轴、纵轴的界限以及标注

在图像中，不能想当然地认为横轴就是 X 轴，纵轴就是 Y 轴。图形因数据不同，纵/横轴标签往往也会不同。这也体现了给纵/横轴设置标签说明的重要性：

```
#设置横轴标签
plt.xlabel("X")
#设置纵轴标签
plt.ylabel("Y")

plt.show()
```

很多时候，需要设置横轴和纵轴的界限，从而得到更加清晰明了的图形：

```
plt.xlim(X.min() * 1.1, X.max() * 1.1)
plt.ylim(C.min() * 1.1, C.max() * 1.1)
```

上述代码将横轴、纵轴的上下限分别设为了数据上下限的 1.1 倍。

此外，为了更好地表示横轴和纵轴数据的含义，可以通过 ticks()函数对横轴和纵轴的含义进行设置和定制。在 Matplotlib 中，图形默认设置的刻度由曲线以及窗口的像素点等因素决定。如果这些刻度的精确度无法满足需求，则需要用户手动添加刻度。

```
plt.xlim(x.min() * 1.1, x.max() * 1.1)
#设置横轴精准刻度
plt.xticks(np.arange(-1, 6, 0.5))

plt.ylim(C.min() * 1.1,C.max() * 1.1)
#设置纵轴精准刻度
plt.yticks(np.arange(-2, 10))

plt.show()
```

xticks() 和 yticks() 需要传入一个列表作为参数。该方法默认使用列表的值来设置刻度标签，如果用户想重新设置刻度标签，则需要传入两个列表参数给 xticks() 和 yticks()，第一个列表的值代表刻度；第二个列表的值代表刻度所显示的标签。

```
#设置横轴精准刻度
plt.xticks(np.arange(-1, 7),
           ["-1m", "0m", "1m", "2m", "3m", "4m", "5m", "6m"])
#设置纵轴精准刻度
plt.yticks(np.arange(-2, 11,2),
           ["-2m", "0m", "2m", "4m", "6m", "8m", "10m"])
plt.show()
```

### 2. 添加图例

如果需要在图的左上角添加一个图例，只需在 plot()函数里以“键-值”的形式增加一个参数。首先需要在绘制曲线的时候增加一个 label 参数，然后再调用 plt.legend() 绘制出一个图例。plt.legend()需要传入一个位置值 loc，其可选的值如表 8.3 所示。

**表 8.3　位置值 loc 的可选值**

| 位置字符串 | 位置代码 | 位置字符串 | 位置代码 |
|---|---|---|---|
| 'best' | 0 | 'center left' | 6 |
| 'upper right' | 1 | 'center right' | 7 |
| 'upper left' | 2 | 'lower center' | 8 |
| 'lower left' | 3 | 'upper center' | 9 |
| 'lower right' | 4 | 'center' | 10 |
| 'right' | 5 | | |

例如：

```
plt.plot(x, y1, color="blue", linewidth=1.0, linestyle="-", label="y1")
plt.plot(x, y2, color="#800080", linewidth=2.0, linestyle="--", label="y2")
plt.legend(loc="upper left")        #将图例绘制在左上角
```

上述代码在图 8.11 的基础上增加了图例，如图 8.14 所示。

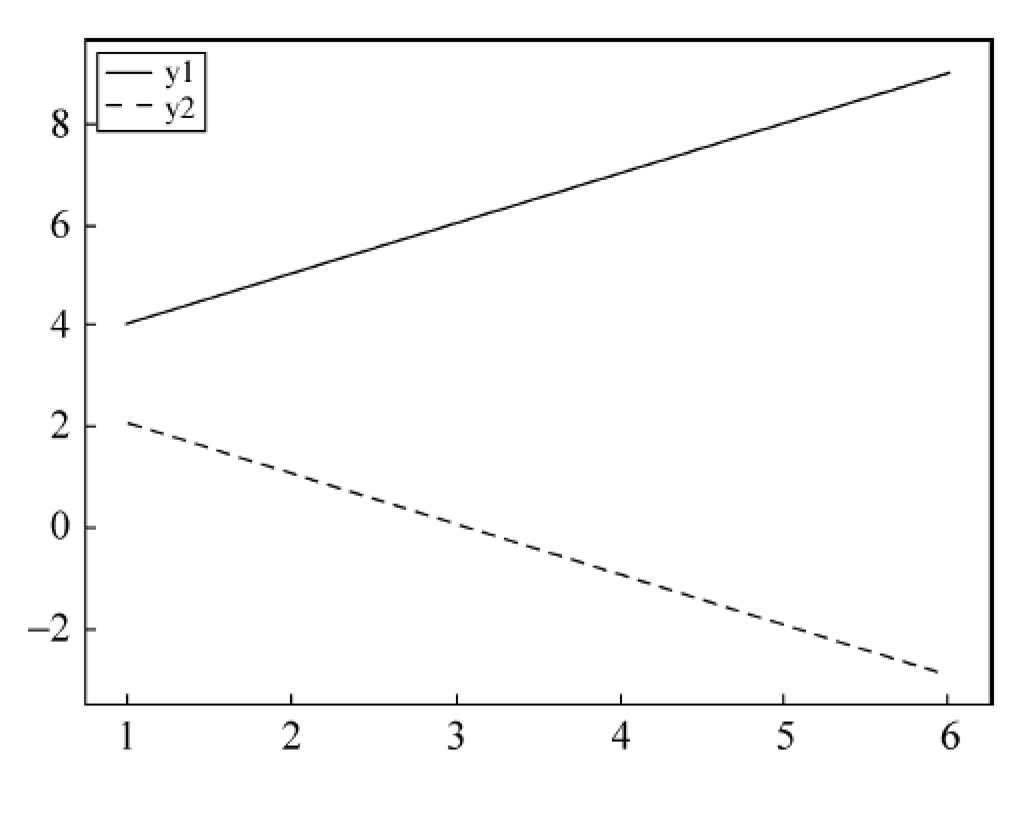



图 8.14　在绘图中添加图例

### 3. 注释特殊点位

有时某些数据点非常关键，需要突显出来，那么可以将该点绘制出来，即绘制散点图，再对其做注释。这里要用到 scatter()和 annotate()函数。

scatter()函数用于绘制散点图，该函数需要传入两个列表作为参数 x 和 y。x 代表要标注点的横轴位置，y 代表要标注点的纵轴位置。x 和 y 列表中下标相同的数据是对应的。例如 x 为[3, 4]，y 为[6, 8]，这表示会绘制点(3,6)、(4,8)。因此，x 和 y 长度要一样。

annotate()函数同样也有两个必传参数，一个是标注内容，另一个是 xy 坐标。标注内容是一个字符串，xy 表示要在哪个位置(点)显示标注内容。xy 的坐标位置一般是在 scatter() 绘制点附近，例如点的右侧或下方。

```
plt.plot(x, y1, color = "blue", linewidth = 1.0, linestyle = " - ", label = "y1")
# 绘制散点(3, 6)
plt.scatter([3], [6], s = 30, color = "blue")        # s 为点的大小
# 对(3, 6)做标注
plt.annotate("(3, 6)",
             xy = (3.3, 5.5),                        # 在(3.3, 5.5)上做标注
             fontsize = 16)                          # 设置字体大小为 16

plt.plot(x, y2, color = "#800080", linewidth = 2.0, linestyle = " -- ", label = "y2")
# 绘制散点(3, 0)
plt.scatter([3], [0], s = 50, color = "#800080")
# 对(3, 0)做标注
```

```
plt.annotate("(3, 0)",
             xy = (3.3, 0),          # 在(3.3, 0)上做标注
             fontsize = 16)          # 设置字体大小为 16
```

上述代码分别在(3,6)和(3,0)两个位置绘制了两个点，如图 8.15 所示。

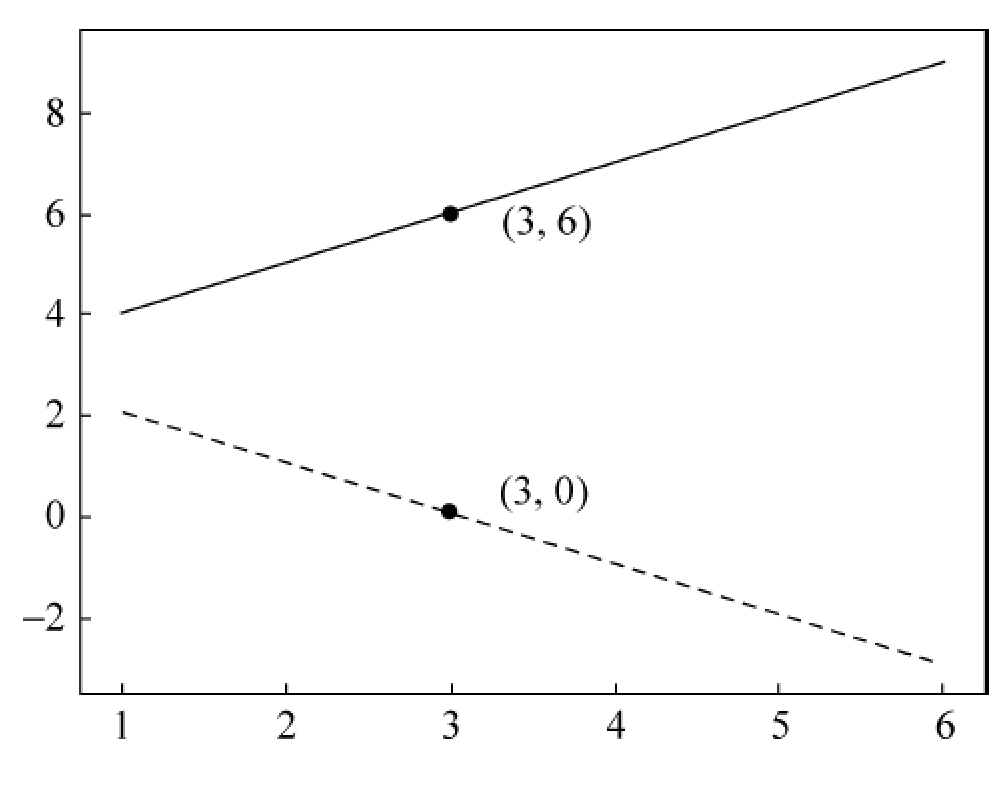



图 8.15　在图表中绘制点

现在点已经被标注出来了，如果还想给点添加注释，需要使用 text()函数。text(x,y,s)的作用是在坐标(x,y) 处添加文本。

```
# 绘制散点(3, 0)
plt.scatter([3], [0], s = 50, color = "#800080")
# 对(3, 0)做标注
plt.annotate("(3, 0)", xy = (3.3, 0))
plt.text(4, -0.5, "该处为重要点位", fontdict = {'size': 12, 'color': 'green'})
```

效果如图 8.16 所示。

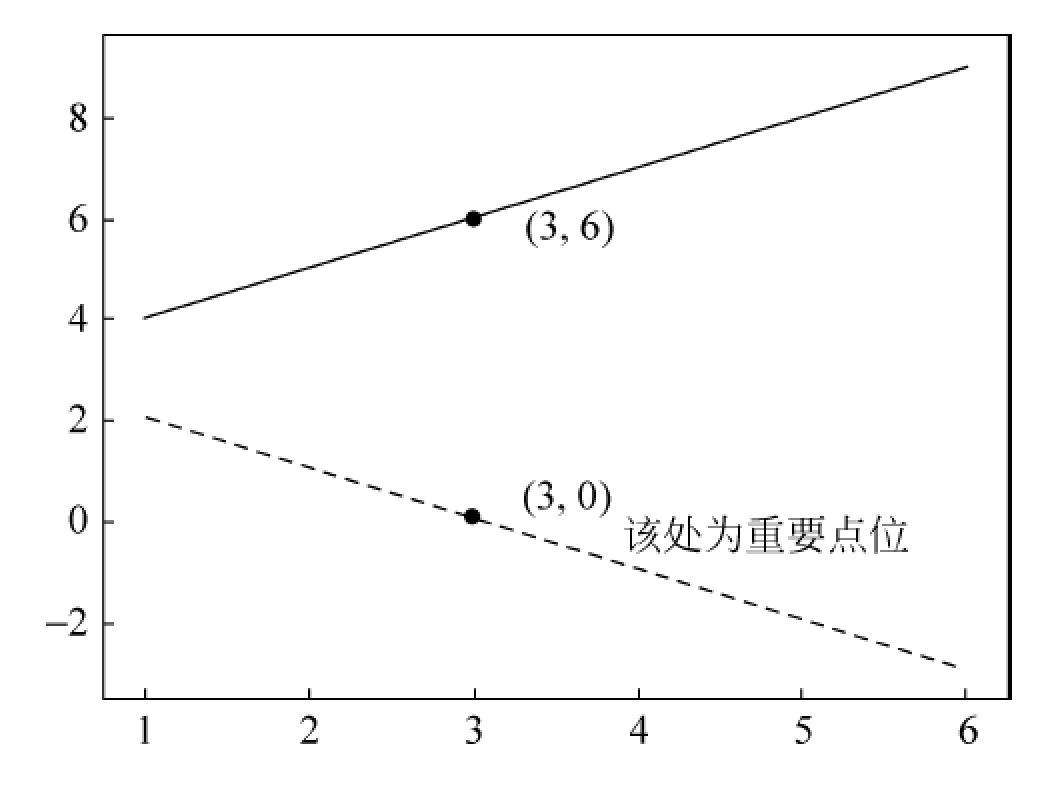



图 8.16　在图表中添加注释文字

### 8.2.4 保存图表

pyplot 的 savefig()函数可以将图表保存成图像文件：

```
plt.savefig('C:\\pic.png',
            dpi = 800,
            bbox_inches = 'tight',
            facecolor = 'w',
            edgecolor = 'blue')
#可支持 png、pdf、svg、ps、eps 等,以后缀名来指定
#dpi 是分辨率
#bbox_inches:图表需要保存的部分. 如果设置为'tight',则尝试剪除图表周围的空白部分
#facecolor,edgecolor: 图像的背景色,默认为'w'(白色)
```

## 8.3 更多高级图表及定制

### 8.3.1 样式

用于 Matplotlib 的样式与用于 HTML 页面的 CSS(层叠样式表)非常相似。在 Matplotlib 中可以使用 for 循环对各个元素的样式逐个修改,使代码量降低,也可以在 Matplotlib 中利用这些样式。

样式表是将自定义样式写入文件,之后如果想要使用这些样式,可以将这些样式导入。这样,就不必为每个图表编写大量自定义的样式代码,而是复用之前写好的样式即可。

导入样式的代码如下：

```
from matplotlib import style
```

接下来,需要指定要使用的样式。Matplotlib 内置了若干可用样式,可以用 style.use()函数来指定：

```
style.use('ggplot')
```

ggplot 的样式大概如图 8.17 所示，可以看到，标签的颜色是灰色的，Axes 的背景是浅灰色。网格实际上是一个白色的实线。

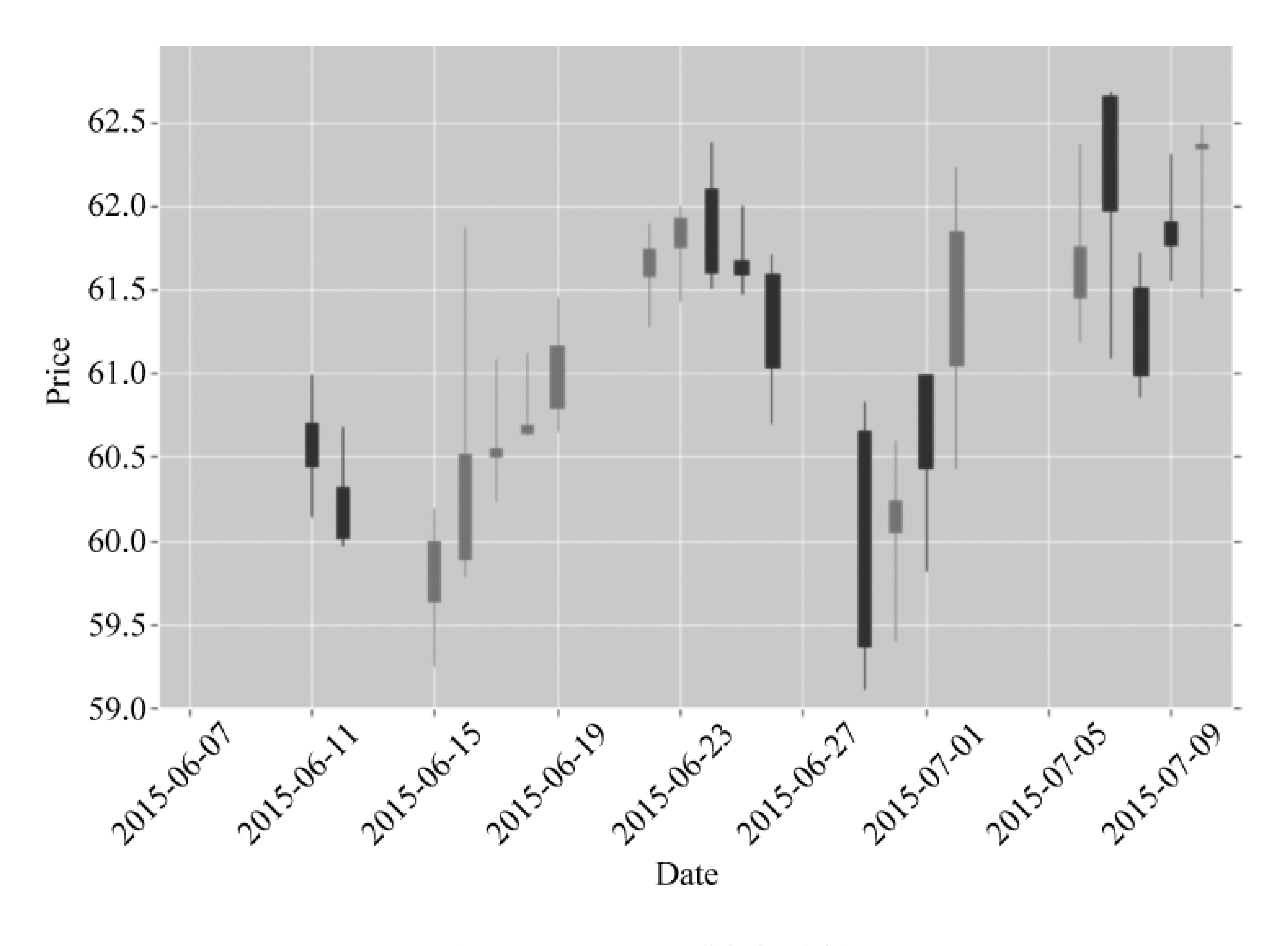



图 8.17　ggplot 样式示例

fivethirtyeight 是另一种内置样式，效果如图 8.18 所示。

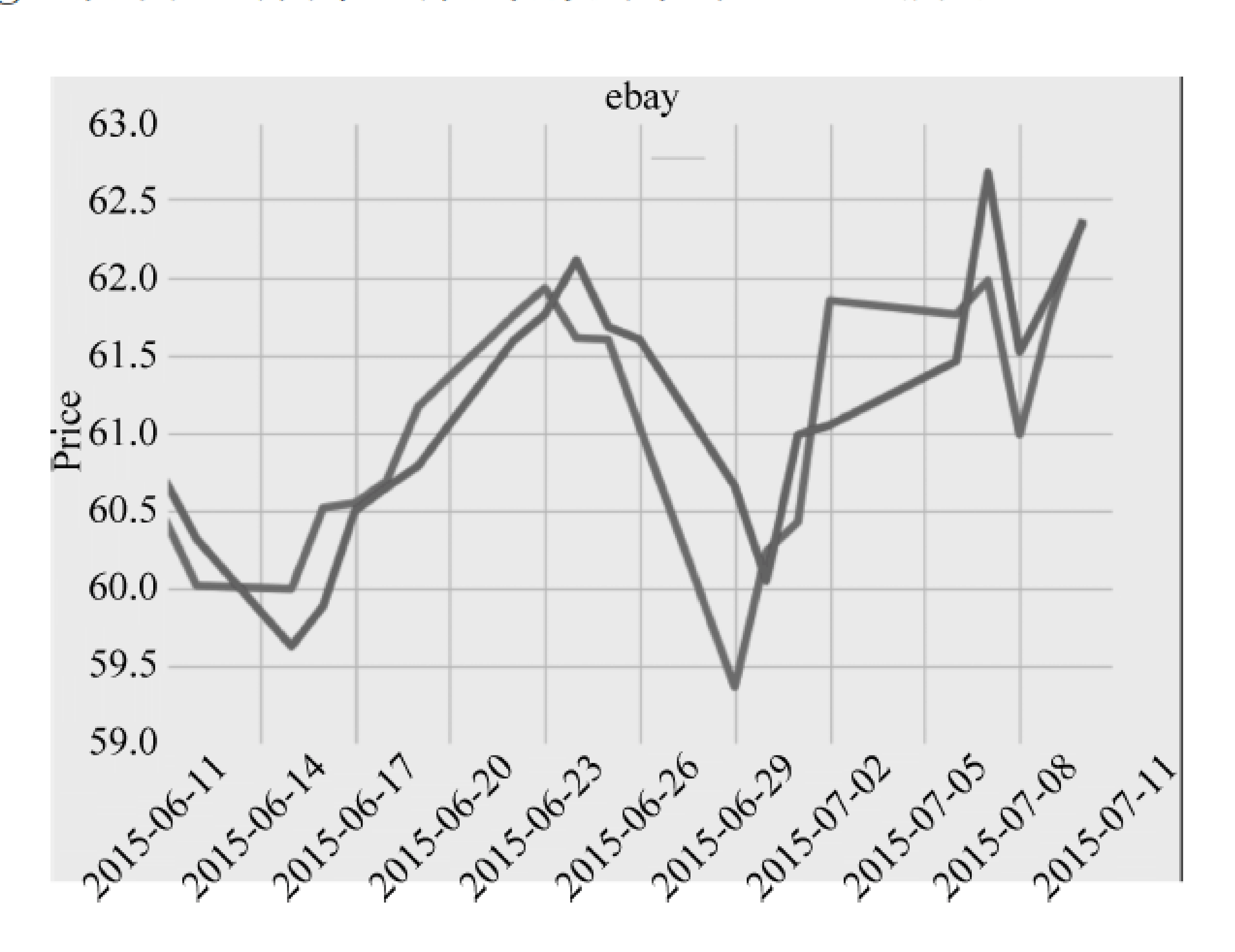



图 8.18　fivethirtyeight 样式示例

如果用户不清楚自己的环境中有哪些内置样式，可以使用下列代码查看所有可用样式：

```
print(plt.style.available)
```

一般来说会输出['bmh', 'dark_background', 'ggplot','fivethirtyeight','grayscale']几种样式。请尝试使用其他几种样式查看结果。

## 8.3.2 subplot 子区


视频讲解

在 Matplotlib 中可以组合许多小图，放在一张大图里面显示，每个小图被称为一个子图。用户可以使用 plt.subplot()实现这样的功能。

例如 plt.subplot(2,2,1)表示将整个图像窗口分为 2 行 2 列，当前位置为 1。使用 plt.plot([0,1],[0,1])在第 1 个位置创建一个小图。

```
plt.subplot(2,2,1)
plt.plot([0,1],[0,1])
```

plt.subplot(2,2,2)表示将整个图像窗口分为 2 行 2 列，当前位置为 2。使用 plt.plot([0,1],[0,2])在第 2 个位置创建一个小图。

```
plt.subplot(2,2,2)
plt.plot([0,1],[0,2])
```

plt.subplot(2,2,3)表示将整个图像窗口分为 2 行 2 列，当前位置为 3。plt.subplot(2,2,3)可以简写成 plt.subplot(223)，Matplotlib 同样可以识别。使用 plt.plot([0,1],[0,3])在第 3 个位置创建一个小图。

```
plt.subplot(223)
plt.plot([0,1],[0,3])
```

plt.subplot(224)表示将整个图像窗口分为 2 行 2 列，当前位置为 4。使用 plt.plot([0,1],[0,4])在第 4 个位置创建一个小图。

```
plt.subplot(224)
plt.plot([0,1],[0,4])

plt.show()
```

在上述示例中每个小图的大小完全相同，事实上小图的大小可以不同。以上面的 4 个小图为例(如图 8.19 所示)，把第 1 个小图放到第 1 行，而把剩下的 3 个小图

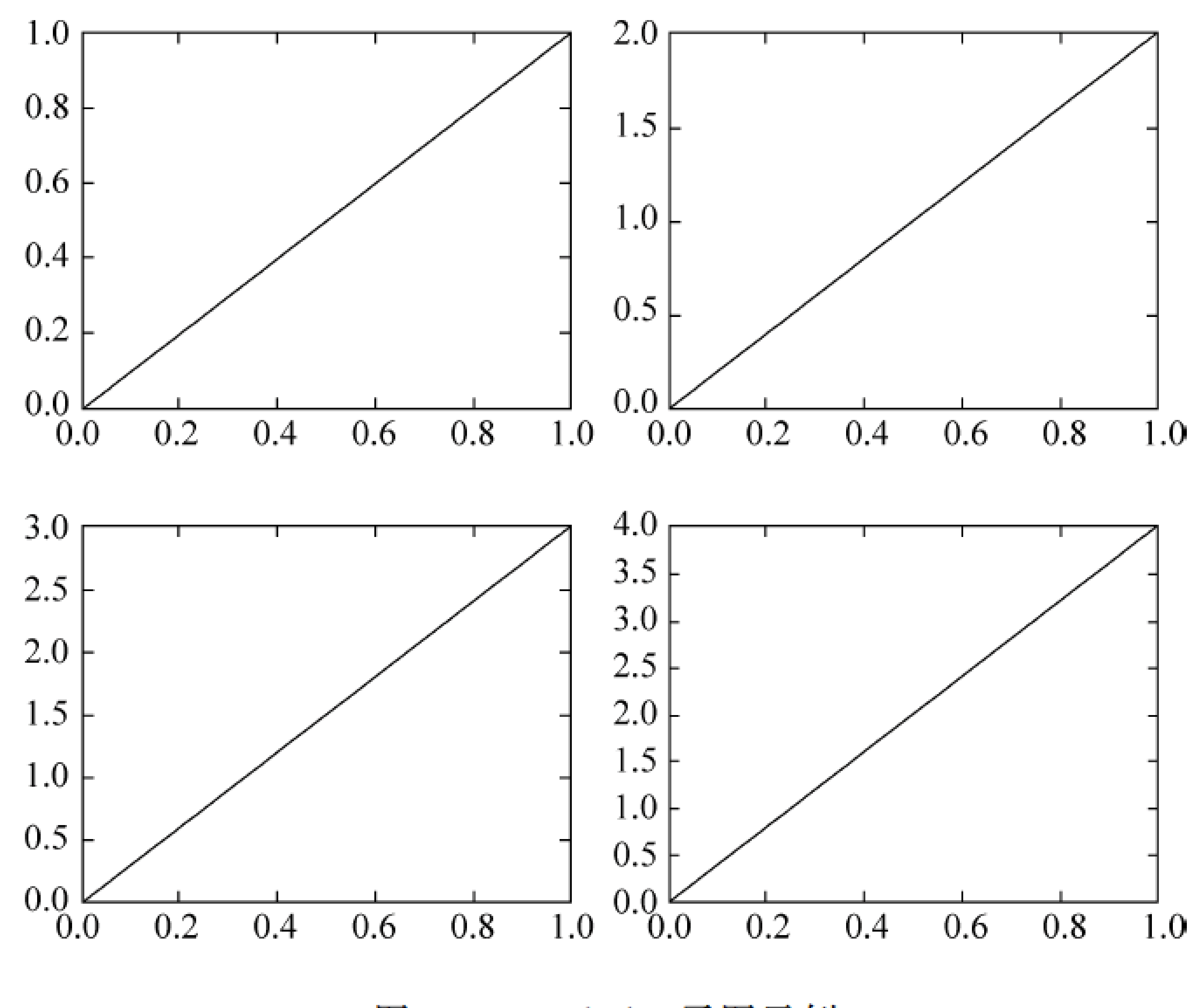



图 8.19 subplot 子图示例

都放到第 2 行。

使用 plt. subplot(2,1,1)将整个图像窗口分为 2 行 1 列，当前位置为 1。使用 plt. plot([0,1],[0,1])在第 1 个位置创建一个小图。

```
plt.subplot(2,1,1)
plt.plot([0,1],[0,1])
```

使用 plt. subplot(2,3,4)将整个图像窗口分为 2 行 3 列，当前位置为 4。使用 plt. plot([0,1],[0,2])在第 4 个位置创建一个小图。

```
plt.subplot(2,3,4)
plt.plot([0,1],[0,2])
```

这里需要解释一下为什么在第 4 个位置放第 2 个小图。在上一步中使用 plt. subplot(2,1,1)将整个图像窗口分为 2 行 1 列，第 1 个小图占用了第 1 个位置，也就是整个第 1 行。这一步中使用 plt. subplot(2,3,4)将整个图像窗口分为 2 行 3 列，于是整个图像窗口的第 1 行就变成了 3 列，也就是成了 3 个位置，所以第 2 行的第 1 个位置是整个图像窗口的第 4 个位置。

使用 plt. subplot(235)将整个图像窗口分为 2 行 3 列，当前位置为 5。使用 plt. plot([0,1],[0,3])在第 5 个位置创建一个小图。同上，再创建 plt. subplot(236)。

```
plt.subplot(235)
plt.plot([0,1],[0,3])

plt.subplot(236)
plt.plot([0,1],[0,4])

plt.show()  #展示
```

显示效果如图 8.20 所示。

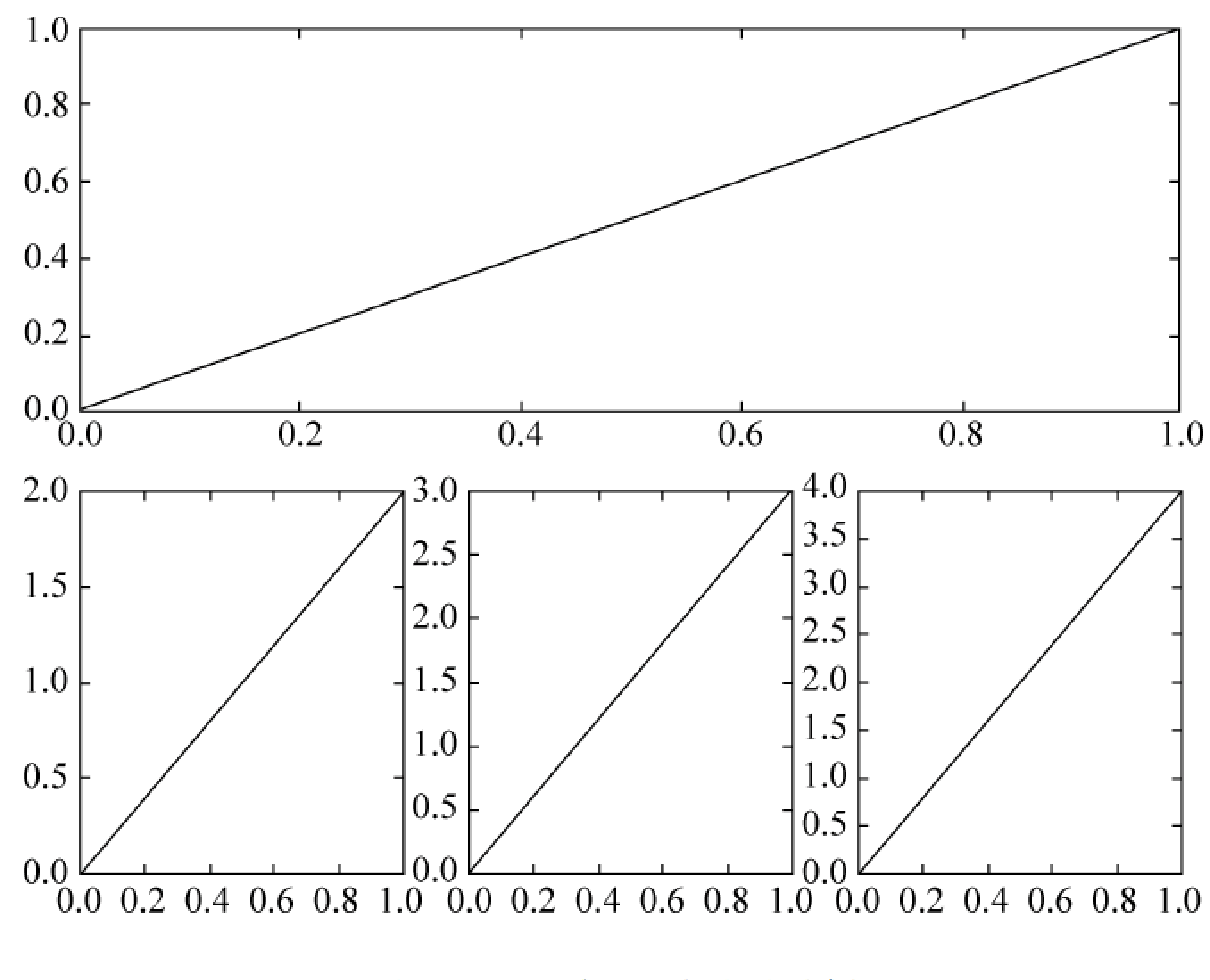



图 8.20　2 行 3 列子图示例

### 8.3.3　图表颜色和填充

颜色和线条填充是可视化图表绘制自定义中的常用方法。

首先是标签的颜色更改，可以通过修改轴中的标签对象来实现：

```
ax1.xaxis.label.set_color('c')
ax1.yaxis.label.set_color('r')
```

通过调用标签对象的 set_color()方法，将标签文本设置为目标颜色。接下来可以为要显示的轴指定具体数字，并做填充。所谓的填充，是指在变量和所选择的一个数值之间填充颜色。例如：

```
ax1.fill_between(date, 0, closep)
for label in ax1.xaxis.get_ticklabels():
```

```
        label.set_rotation(45)
ax1.grid(True) #, color = 'g', linestyle = '-', linewidth = 5)
ax1.xaxis.label.set_color('c')
ax1.yaxis.label.set_color('r')
ax1.set_yticks([0,25,50,75])
plt.xlabel('Date')
plt.ylabel('Price')
plt.title(stock)
plt.legend()
plt.subplots_adjust(left = 0.09, bottom = 0.20, right = 0.94, top = 0.90, wspace = 0.2,
hspace = 0)
plt.show()
```

显示结果如图 8.21 所示。

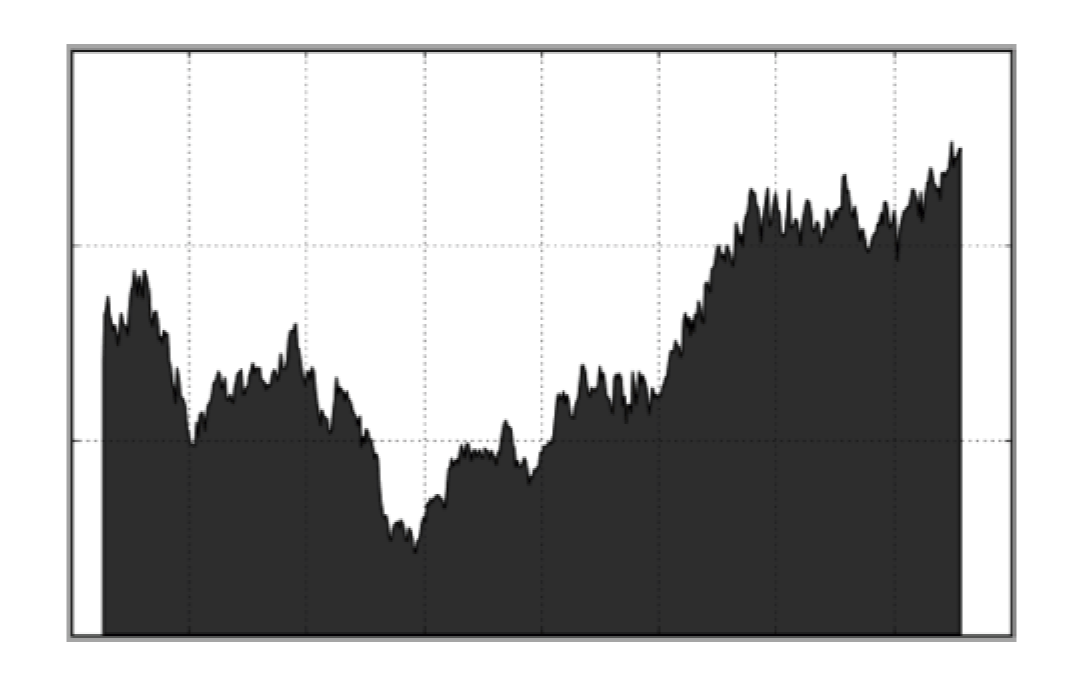


图 8.21　填充图表示例

## 8.3.4　动画

可以使用 Matplotlib 实现简单的动画功能，function animation 是其中一种较为简便的方法，具体可参考 Matplotlib animation API。首先需要引入 pyplot 和 animation 两个模块：

```
from matplotlib import pyplot as plt
from matplotlib import animation
import numpy as np
fig, ax = plt.subplots()
```

所用数据是一个 0～2π 的正弦曲线：

```
x = np.arange(0, 2 * np.pi, 0.01)
line, = ax.plot(x, np.sin(x))
```

接着构造自定义动画函数 animate(),用来更新每一帧上各个 x 对应的 y 坐标值,参数表示第 i 帧:

```
def animate(i):
    line.set_ydata(np.sin(x + i/10.0))
    return line,
```

然后构造开始帧函数 init():

```
definit():
    line.set_ydata(np.sin(x))
    return line,
```

接下来调用 FuncAnimation()函数生成动画。参数说明如下。

- fig:进行动画绘制的 figure。
- func:自定义动画函数,即传入刚定义的函数 animate。
- frames:动画长度,一次循环包含的帧数。
- init_func:自定义开始帧,即传入刚定义的函数 init。
- interval:更新频率,以 ms 计。
- blit:选择更新所有点,还是仅更新产生变化的点。通常选择 True,但 Mac 用户请选择 False,否则无法显示动画。

```
ani = animation.FuncAnimation(fig = fig,
                              func = animate,
                              frames = 100,
                              init_func = init,
                              interval = 20,
                              blit = False)
```

显示动画:

```
plt.show()
```

当然,也可以将动画以 MP4 格式保存下来,但首先要保证环境中已经安装了 FFmpeg 或者 Mencoder。

```
ani.save('basic_animation.mp4', fps = 30, extra_args = ['-vcodec', 'libx264'])
```

## 本章小结

本章先向读者介绍说明数据可视化的基本概念和常用的数据图表；然后介绍Python的Matplotlib可视化库，重点介绍如何使用该库的pyplot对象制作图表，包括点线图、柱状图等常用图表，接下来介绍图表定制的基本方法，最后讲述高级的图表定制。

## 习题

1. 创建一个函数，要求绘制出y=x^a幂函数的图形，该函数输入一个数值型参数作为幂函数的指数。例如输入参数2，则绘制出y=x^2的图形。

2. 随机生成1000个[0,1]区间内的数字作为X，再随机生成1000个[0,1]区间内的数字作为Y，将1000个X、Y坐标绘制成散点图。

# 第 9 章



# 数据库应用开发

**本章学习目标：**

- 理解数据库的基本概念
- 熟练掌握常用的 Python 数据库管理的相关库
- 熟练掌握使用 Python 进行数据库的操作

本章先向读者介绍数据库的基本概念和 Python 的数据库开发环境；然后介绍嵌入式数据库、关系型数据库、NoSQL 数据库的基本概念，并分别以 SQLite、MySQL 和 MongoDB 为例，介绍其使用方法和利用 Python 对其进行操作的基本方法。

## 9.1 Python 与数据库

### 9.1.1 数据库简介

当程序运行的时候，数据都存在于内存当中。当程序终止的时候，通常需要将数据保存到磁盘上。无论是保存到本地磁盘，还是通过网络保存到服务器上，最终都会将数据写入磁盘文件。

如何定义数据的存储格式是一个大问题。如果用户自己定义存储格式，例如保

存一个班级中所有学生成绩的成绩单，如表 9.1 所示。

**表 9.1 成绩单**

| 名　　字 | 成　　绩 |
|---|---|
| 张三 | 99 |
| 李四 | 85 |
| 王五 | 82 |
| 赵六 | 92 |

可以用一个文本文件保存，一行保存一个学生，姓名和成绩之间用逗号“,”隔开：

```
张三,99
李四,85
王五,82
赵六,92
```

这种格式被称为 CSV(Comma-Separated Values，逗号分隔值)。当然，也可以用其他文本格式保存，例如 JSON 格式：

```
[
    {"name":"张三","score":99},
    {"name":"李四","score":85},
    {"name":"王五","score":82},
    {"name":"赵六","score":92}
]
```

上述两种都属于通用格式，用户还可以定义自己的各种保存格式，但是这样存储和读取也需要自己实现，并且每定义一种格式都需要一种存储和读取的方法。另外，对于文本文件的读取不能实现快速查询，只有把数据全部读到内存中才能遍历，但有时候数据的大小远远超过了内存(例如蓝光电影，40GB 的数据)，根本无法一次性将其全部读入。

为了便于程序保存和读取数据，并能直接通过检索条件快速查询到指定的数据，出现了数据库(Database)这种专门用于集中存储和查询的软件。

数据库软件的历史非常久远，早在 1950 年数据库就诞生了。它经历了网状数据库、层次数据库等各种形态，直到 20 世纪 70 年代诞生了基于关系模型的关系数据库，并被广泛使用至今。

关系模型有一套复杂的数学理论，但从概念上是十分容易理解的。举个学校的

例子：

假设某小学有 3 个年级，要表示出这 3 个年级，可以用一个表格画出来，如图 9.1 所示。

每个年级又有若干个班，要把所有班级表示出来，可以再画一个表格，如图 9.2 所示。

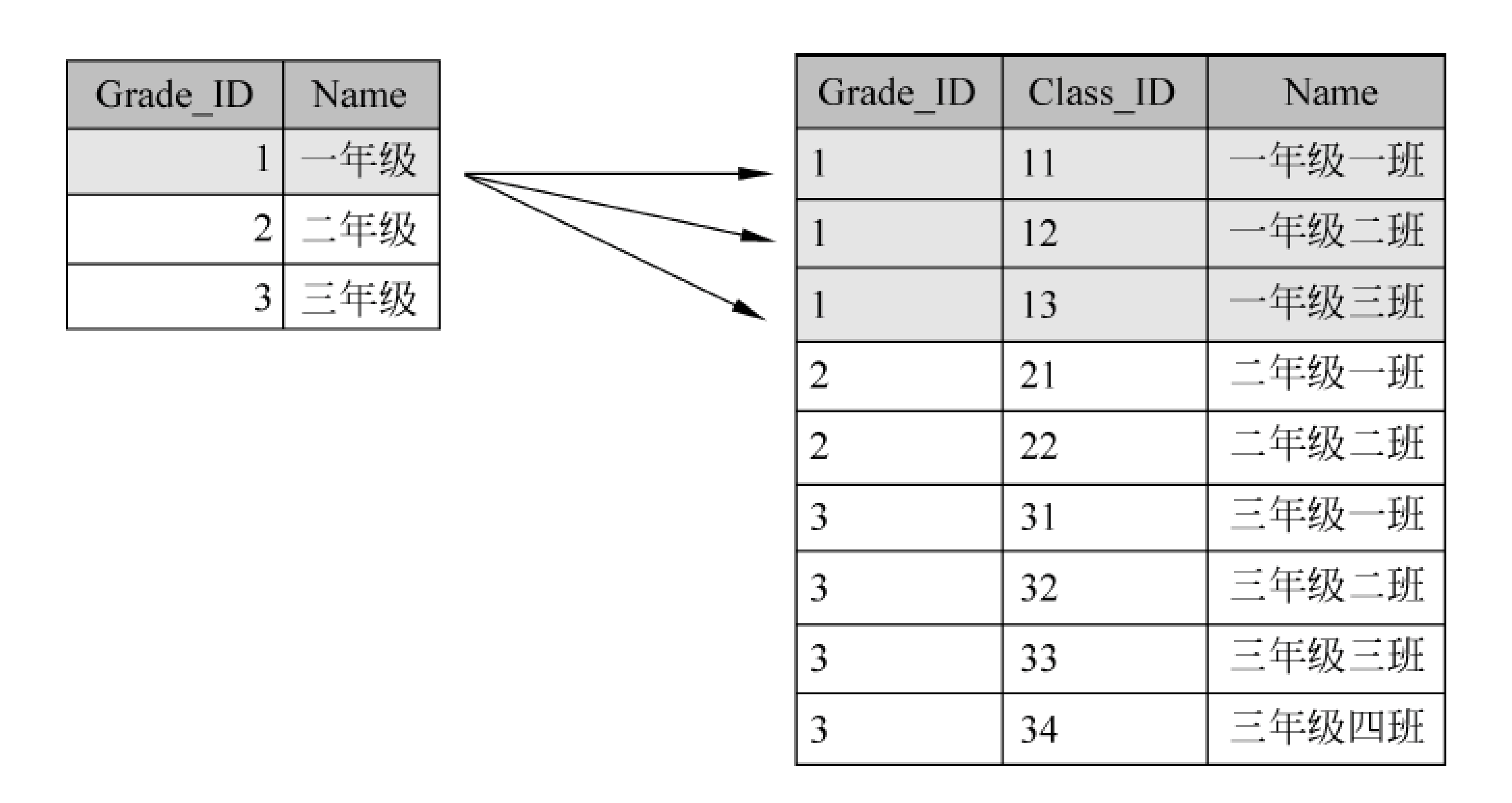
| Grade_ID | Name |
| --- | --- |
| 1 | 一年级 |
| 2 | 二年级 |
| 3 | 三年级 |

图 9.1　用表格表示年级

| Grade_ID | Class_ID | Name |
| --- | --- | --- |
| 1 | 11 | 一年级一班 |
| 1 | 12 | 一年级二班 |
| 1 | 13 | 一年级三班 |
| 2 | 21 | 二年级一班 |
| 2 | 22 | 二年级二班 |
| 3 | 31 | 三年级一班 |
| 3 | 32 | 三年级二班 |
| 3 | 33 | 三年级三班 |
| 3 | 34 | 三年级四班 |

图 9.2　用表格表示班级

这两个表格有个映射关系，就是根据 Grade_ID 可以在班级表中查找到对应的所有班级，如图 9.3 所示。



| Grade_ID | Name |
| --- | --- |
| 1 | 一年级 |
| 2 | 二年级 |
| 3 | 三年级 |

| Grade_ID | Class_ID | Name |
| --- | --- | --- |
| 1 | 11 | 一年级一班 |
| 1 | 12 | 一年级二班 |
| 1 | 13 | 一年级三班 |
| 2 | 21 | 二年级一班 |
| 2 | 22 | 二年级二班 |
| 3 | 31 | 三年级一班 |
| 3 | 32 | 三年级二班 |
| 3 | 33 | 三年级三班 |
| 3 | 34 | 三年级四班 |

图 9.3　根据 Grade_ID 在班级表中查找到对应的所有班级

也就是 Grade 表的每一行对应 Class 表的多行。在关系数据库中，这种基于表(Table)的一对多的关系就是关系数据库的基础。

根据某个年级的 ID 就可以查找该年级所有班级的记录，这种查询语句在关系数

据库中称为 SQL 语句，可以写成：

```
SELECT * FROM Class WHERE Grade_ID = '1';
```

其结果也是一个表：

```
-----------+----------+----------
Grade_ID | Class_ID | Name
-----------+----------+----------
1          | 11         | 一年级一班
-----------+----------+----------
1          | 12         | 一年级二班
-----------+----------+----------
1          | 13         | 一年级三班
-----------+----------+----------
```

类似地，Class 表的一行记录又可以关联到 Student 表的多行记录，如图 9.4 所示。

| Grade_ID | Class_ID | Name |
|---|---|---|
| 1 | 11 | 一年级一班 |
| 1 | 12 | 一年级二班 |
| 1 | 13 | 一年级三班 |
| 2 | 21 | 二年级一班 |
| 2 | 22 | 二年级二班 |
| 3 | 31 | 三年级一班 |
| 3 | 32 | 三年级二班 |
| 3 | 33 | 三年级三班 |
| 3 | 34 | 三年级四班 |

| Class_ID | Num | Name | Score |
|---|---|---|---|
| 11 | 10001 | Michael | 99 |
| 11 | 10002 | BOb | 85 |
| 11 | 10003 | Bart | 59 |
| 11 | 10004 | Lisa | 87 |
| 12 | 10005 | Tracy | 91 |
| … | | … | … |

图 9.4　Class 表的一行记录关联 Student 表的多行记录

总的来说，关系数据由二维表格和表格之间的联系作为基础，再搭配一系列的工具，例如查询、索引、视图、存储过程等，能够实现极其复杂的数据管理功能。

下面介绍几种目前广泛使用的关系数据库。

(1) 付费的商用数据库。

- Oracle：世界上最流行、最专业的商业关系型数据库系统。
- SQL Server：微软的产品，专为 Windows 定制。
- DB2：IBM 的产品，主要应用于大型应用系统。

这些数据库都是不开源而且付费的，最大的优势是售后支持服务出了问题可以

找厂家解决。

(2) 免费的开源数据库。

- MySQL：最主流的开源数据库。
- PostgreSQL：功能、性能都很优秀的开源数据库，知名度略低于 MySQL，但发展势头迅猛。
- SQLite：嵌入式数据库，适合桌面和移动应用。

在如今以互联网为基础的大数据时代，经常需要部署成千上万的数据库服务器，所以无论是 Google、Facebook 还是国内的百度、阿里巴巴、腾讯等互联网巨头，无一例外都选择了免费的开源数据库作为主要的数据存储方案。

对于初学者来说，建议至少熟练掌握 MySQL 和 PostgreSQL 数据库中的一种，一方面，这两种数据库可以免费下载，无须付费；另一方面，这二者与商业数据库相比更为小巧，操作管理更为简便，学习难度更低。

当然，关系型数据库虽然是主流，但不是唯一的数据库类型。最近几年兴起的非关系型数据库正在日益发展并逐渐挑战关系型数据库的主流地位，很多 NoSQL 产品宣传其速度和规模远远超过关系型数据库。

NoSQL 一词最早出现于 1998 年，它是 Carlo Strozzi 开发的一个轻量、开源、不提供 SQL 功能的关系数据库。

2009 年，Last. fm 的 Johan Oskarsson 发起了一次关于分布式开源数据库的讨论，来自 Rackspace 的 Eric Evans 再次提出了 NoSQL 的概念，这时的 NoSQL 主要指非关系型、分布式、不提供 ACID 的数据库设计模式。

2009 年在亚特兰大举行的"no:sql(east)"讨论会是一个里程碑，其口号是"SELECT fun, profit FROM real_world WHERE relational = False;"。因此，对 NoSQL 最普遍的解释是"非关联型的"，强调 Key-Value Stores 和文档数据库的优点，而不是单纯地反对关系型数据库。

现在一般认为 NoSQL 的优点主要包括高可扩展性、分布式计算、低成本、架构的灵活性、半结构化数据、没有复杂的关系。当然，NoSQL 也有一些缺点，例如没有标准化、有限的查询功能(到目前为止)、缺少优秀的客户端程序等。

### 9.1.2 Python 数据库工作环境

由于各数据库之间的应用接口非常混乱，实现各不相同，如果项目需要更换数据

库，则需要做大量的修改，非常不便。Python DB-API 的出现就是为了解决这样的问题。

Python 所有的数据库接口程序都在一定程度上遵守 Python DB-API 规范。DB-API 定义了一系列必需的对象和数据库存取方式，以便为各种底层数据库系统和多种多样的数据库接口程序提供一致的访问接口。由于 DB-API 为不同的数据库提供了一致的访问接口，在不同的数据库之间移植代码成为一件轻松的事情。

Python DB-API 的使用流程如下：

(1) 引入 API 模块。

(2) 获取与数据库的连接。

(3) 执行 SQL 语句和存储过程。

(4) 关闭数据库连接。

### 1. 使用 connect()创建 Connection 连接

DB-API 中的 connect()方法能生成一个 Connect 对象，用户可以通过这个对象来访问数据库。符合标准的模块都会实现 connect()方法。

connect()方法的参数如下。

- user：数据库连接用户名。
- password：连接密码。
- host：主机名。
- database：数据库名。
- dsn：数据源名。

数据库连接参数可以以一个 DSN 字符串的形式提供，例如 connect (dsn='host:MYDB',user='root',password='')。当然，不同的数据库接口程序可能有些差异，并非都是严格按照规范实现，例如 MySQLdb 使用 db 参数而不是使用规范推荐的 database 参数来表示要访问的数据库。

此外，Connect 对象还有如下方法。

- close()：关闭当前 Connect 对象，关闭后无法进行操作，除非再次创建连接。
- commit()：提交当前事务，如果是支持事务的数据库执行增/删/改后没有 commit，则数据库默认回滚。
- rollback()：取消当前事务。

- cursor()：创建游标对象。

### 2. 使用 Cursor()创建游标对象

Cursor(游标)对象有如下属性和方法。

1) 常用方法

- close()：关闭此游标对象。
- fetchone()：得到结果集的下一行。
- fetchmany([size=cursor.arraysize])：得到结果集的下几行。
- fetchall()：得到结果集中剩下的所有行。
- excute(sql[，args])：执行一个数据库查询或命令。
- excutemany(sql，args)：执行多个数据库查询或命令。

2) 常用属性

- connection：创建此游标对象的数据库连接。
- arraysize：使用 fetchmany()方法一次取出多少条记录，默认为 1。
- lastrowid：上一行的行号。

3) 其他方法

- __iter__()：创建一个可迭代对象(可选)。
- next()：获取结果集的下一行(如果支持迭代)。
- nextset()：移到下一个结果集(如果支持)。
- callproc(func[,args])：调用一个存储过程。
- setinputsizes(sizes)：设置输入最大值(必须有，但具体实现是可选的)。
- setoutputsizes(sizes[,col])：设置大列获取的最大缓冲区大小。

4) 其他属性

- description：返回游标活动状态(包含 7 个元素，即 name、type_code、display_size、internal_size、precision、scale、null_ok)的元组，只有 name 和 type_code 是必需的。
- rowcount：最近一次 execute()创建或影响的行数。
- messages：游标执行后数据库返回的信息元组(可选)。
- rownumber：当前结果集中游标所在行的索引(起始行号为 0)。

## 9.2　本地数据库 SQLite



视频讲解

### 9.2.1　SQLite 简介

SQLite 是一种嵌入式数据库，它的数据库就是一个文件。由于 SQLite 本身是用 C 语言写的，而且体积很小，所以经常被集成到各种应用程序中，甚至在 iOS 和 Android 的 App 中都可以集成。

市面上主流的数据库很多，为什么要使用 SQLite 呢？简单来说，SQLite 有下面几个优势：

(1) SQLite 不需要一个单独的服务器进程或系统操作(服务器)。

(2) SQLite 不需要配置，这意味着它不需要安装或管理。

(3) 一个完整的 SQLite 数据库可存储在跨平台的磁盘文件中。

(4) SQLite 非常小、重量轻，完全配置的版本小于 400KB，省略可选功能的版本甚至小于 250KB。

(5) SQLite 是自配置的、独立的，这意味着它不需要依赖任何外部应用程序或环境。

(6) SQLite 的交易完全符合 ACID，允许多个进程或线程安全访问。

(7) SQLite 支持大多数(SQL2)符合 SQL92 标准的查询语言功能。

(8) SQLite 提供了简单和易于使用的 API。

(9) SQLite 可在 UNIX 和 Windows 中运行。

当然，SQLite 也不是没有缺点，它一般只能用于处理小到中型数据量的存储，对于高并发、高流量的应用并不适用。

### 9.2.2　Python 内置的 sqlite3 模块

Python 本身内置了 sqlite3，所以在 Python 中使用 SQLite 甚至不需要安装任何软件即可直接使用，这也体现了 SQLite 的优势。

在使用 SQLite 之前，用户先要搞清楚几个概念。

表是数据库中存放关系数据的集合，在一个数据库中通常包含多个表，例如学生

表、班级表、学校表等。表和表之间通过外键关联。

如果要操作关系数据库，首先需要连接到数据库，一个数据库连接称为Connection。

在连接到数据库之后，需要打开游标（或称之为Cursor）通过Cursor执行SQL语句，然后获得执行结果。

SQLite的驱动内置在Python标准库中，可以直接操作SQLite数据库。

下列代码实现了创建连接、创建表、插入记录、关闭连接等数据库基本操作：

```
#导入SQLite驱动
>>> import sqlite3
#连接到SQLite数据库
#数据库文件是test.db
#如果文件不存在,会自动在当前目录创建
>>> conn = sqlite3.connect('test.db')
#创建一个Cursor
>>> cursor = conn.cursor()
#执行一条SQL语句,创建user表
>>> cursor.execute('CREATE TABLE user (id VARCHAR(20) PRIMARY KEY, name VARCHAR(20))')
<sqlite3.Cursor object at 0x10f8aa260>
#继续执行一条SQL语句,插入一条记录
>>> cursor.execute('INSERT INTO user (id, name) VALUES (\'1\', \'Michael\')')
<sqlite3.Cursor object at 0x10f8aa260>
#通过rowcount获得插入的行数
>>> cursor.rowcount
1
#关闭Cursor
>>> cursor.close()
#提交事务
>>> conn.commit()
#关闭Connection
>>> conn.close()
```

下列代码实现了使用Python对SQLite进行记录查询：

```
>>> conn = sqlite3.connect('test.db')
>>> cursor = conn.cursor()
#执行查询语句
>>> cursor.execute('SELECT * FROM user WHERE id=?', ('1',))
<sqlite3.Cursor object at 0x10f8aa340>
#获得查询结果集
>>> values = cursor.fetchall()
>>> values
[('1', 'Michael')]
```

```
>>> cursor.close()
>>> conn.close()
```

在使用 Python 的 DB-API 时要注意分辨 Connection 和 Cursor 对象的区别，打开后一定要记得关闭。

在使用 Cursor 对象执行 INSERT、UPDATE、DELETE 语句时，执行结果由 rowcount 返回影响的行数。

在使用 Cursor 对象执行 SELECT 语句时，通过 fetchall()可以获取结果集。结果集是一个 list，每个元素都是一个 tuple，对应一行记录。

如果 SQL 语句带有参数，那么需要把参数按照位置传递给 execute()方法，有几个“?”占位符就必须对应几个参数。例如：

```
cursor.execute('SELECT * FROM user WHERE name = ? AND pwd = ?', ('abc', 'password'))
```

## 9.3 关系型数据库



视频讲解

### 9.3.1 关系型数据库基本操作与 SQL

关系型数据库是一个数据集合，保存了很多表。“关系”就是指各个表之间的关联。

对表和表中的数据进行可视化是很有必要的，一般把表显示为由行、列组成的表格。每一行表示一条记录，每一列表示一个字段。行头是字段名，其余行是数据。

SQL 是维护以及使用关系型数据库中数据的一种标准的计算机语言，简单地说就是用户用来和关系型数据库交互的语言。SQL 与其他计算机语言(例如 C、Java、C# 等)不同，SQL 是一种声明式的语言，它经常使用一条单独的语句来声明预期的目标。需要注意的是，SQL 只关注关系型数据库系统，而不是整个计算机系统。

需要在数据库上执行的大部分工作都由 SQL 语句完成，例如下面的语句从 persons 表中选取 LastName 列的数据：

```
SELECT LastName FROM Persons
```

可以把 SQL 分为两个部分，即数据操作语言(DML)和数据定义语言(DDL)。

SQL 是用于执行查询的语言，但是 SQL 语言也包含用于更新、插入和删除记录的语法。

查询和更新指令构成了 SQL 的 DML 部分。

- SELECT：从数据库表中获取数据。
- UPDATE：更新数据库表中的数据。
- DELETE：从数据库表中删除数据。
- INSERT INTO：向数据库表中插入数据。

SQL 的数据定义语言(DDL)部分使用户能够创建或删除表格。用户也可以定义索引(键)，规定表之间的连接，以及施加表间的约束。

SQL 中最重要的 DDL 语句如下。

- CREATE DATABASE：创建新数据库。
- ALTER DATABASE：修改数据库。
- CREATE TABLE：创建新表。
- ALTER TABLE：变更(改变)数据库表。
- DROP TABLE：删除表。
- CREATE INDEX：创建索引(搜索键)。
- DROP INDEX：删除索引。

### 9.3.2 操作 MySQL

MySQL 是 Web 世界中使用最广泛的数据库产品。SQLite 的特点是轻量级、可嵌入，但它不能承受高并发访问，适合桌面和移动应用。MySQL 是为服务器端设计的数据库，能承受高并发访问，并且占用的内存远远超过 SQLite。

此外，MySQL 内部有多种数据库引擎，最常用的引擎是支持数据库事务的 InnoDB。

用户可以直接从 MySQL 官方网站下载最新的版本，选择对应的平台下载安装文件进行安装即可。

在 Windows 上安装时请选择 UTF-8 编码，以便正确地处理中文。

由于 MySQL 服务器以独立的进程运行，并通过网络对外服务，所以需要支持 Python 的 MySQL 驱动来连接到 MySQL 服务器。MySQL 官方提供了 mysql-

connector-python 驱动，可使用 pip 命令工具安装：

```
$ pip install mysql-connector-python
```

下面的代码演示如何连接到 MySQL 服务器的 test 数据库：

```
import mysql.connector

# 打开数据库连接
db = mysql.connector.connect(user = 'testuser', password = 'test123', host = '127.0.0.1',
database = 'TESTDB')

# 使用 cursor()方法获取操作游标
cursor = db.cursor()

# 使用 execute()方法执行 SQL 语句
cursor.execute("SELECT VERSION()")

# 使用 fetchone() 方法获取一条数据
data = cursor.fetchone()

print ("Database version : %s " % data)

# 关闭数据库连接
db.close()
```

执行以上代码的输出如下，具体输出内容会因各人所安装数据库版本的差异而有所不同：

```
Database version : 5.7.21
```

如果数据库连接存在，用户可以使用 execute()方法为数据库创建表，如下所示创建 EMPLOYEE 表：

```
import mysql.connector

# 打开数据库连接
db = mysql.connector.connect(user = 'testuser', password = 'test123', host = '127.0.0.1',
database = 'TESTDB')

# 使用 cursor()方法获取操作游标
cursor = db.cursor()

# 如果数据表已经存在，使用 execute()方法删除表
```

```
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

#创建数据表的 SQL 语句
sql = """CREATE TABLE EMPLOYEE (
         FIRST_NAME CHAR(20),
         LAST_NAME CHAR(20),
         AGE INT,
         SEX CHAR(1),
         INCOME FLOAT )"""

cursor.execute(sql)
cursor.close()

#关闭数据库连接
db.close()
```

以下实例通过执行 SQL INSERT 语句向 EMPLOYEE 表中插入记录：

```
import mysql.connector

#打开数据库连接
db = mysql.connector.connect(user = 'testuser', password = 'test123', host = '127.0.0.1',
database = 'TESTDB')

#使用 cursor()方法获取操作游标
cursor = db.cursor()

#SQL 插入语句
sql = """INSERT INTO EMPLOYEE(FIRST_NAME,
         LAST_NAME, AGE, SEX, INCOME)
         LVALUES ('Mac', 'Mohan', 20, 'M', 2000)"""
try:
    #执行 SQL 语句
    cursor.execute(sql)
    #提交到数据库执行
    db.commit()
except:
    #发生错误时回滚
    db.rollback()

#关闭数据库连接
db.close()
```

上述代码的 SQL 也可以写成如下形式：

```
#SQL 插入语句
sql = "INSERT INTO EMPLOYEE(FIRST_NAME,
        LAST_NAME, AGE, SEX, INCOME)
        VALUES ('%s', '%s', '%d', '%c', '%d')" %
        ('Mac', 'Mohan', 20, 'M', 2000)
```

以下代码使用变量向 SQL 语句中传递参数：

```
…
user_id = "test123"
password = "password"

con.execute('INSERT INTO Login VALUES("%s", "%s")' %
               (user_id, password))
…
```

Python 查询 MySQL 使用 fetchone() 方法获取单条数据，使用 fetchall() 方法获取多条数据。

- fetchone()：该方法获取下一个查询结果集，结果集是一个对象。
- fetchall()：接收全部返回结果行。
- rowcount：是一个只读属性，返回执行 execute()方法后影响的行数。

例如查询 EMPLOYEE 表中 salary(工资)字段大于 1000 的所有数据：

```
import mysql.connector

#打开数据库连接
db = mysql.connector.connect(user = 'testuser', password = 'test123', host = '127.0.0.1',
database = 'TESTDB')

#使用 cursor()方法获取操作游标
cursor = db.cursor()

#SQL 查询语句
sql = "SELECT * FROM EMPLOYEE
        WHERE INCOME > '%d'" % (1000)
try:
    #执行 SQL 语句
    cursor.execute(sql)
    #获取所有记录列表
    results = cursor.fetchall()
```

```
    for row in results:
        fname = row[0]
        lname = row[1]
        age = row[2]
        sex = row[3]
        income = row[4]
        #打印结果
        print "fname = %s,lname = %s,age = %d,sex = %s,income = %d" %
                (fname, lname, age, sex, income)
except:
    print "Error: unable to fecth data"
#关闭数据库连接
db.close()
```

以上代码的执行结果如下：

```
fname = Mac, lname = Mohan, age = 20, sex = M, income = 2000
```

更新操作用于更新数据表中的数据，以下实例将 EMPLOYEE 表中 SEX 字段为 'M' 的 AGE 字段递增 1：

```
import mysql.connector

#打开数据库连接
db = mysql.connector.connect(user = 'testuser', password = 'test123', host = '127.0.0.1',
database = 'TESTDB')

#使用 cursor()方法获取操作游标
cursor = db.cursor()

#SQL 更新语句
sql = "UPDATE EMPLOYEE SET AGE = AGE + 1 WHERE SEX = '%c'" % ('M')
try:
    #执行 SQL 语句
    cursor.execute(sql)
    #提交到数据库执行
    db.commit()
except:
    #发生错误时回滚
    db.rollback()

#关闭数据库连接
db.close()
```

删除操作用于删除数据表中的数据，以下实例演示了删除数据表 EMPLOYEE

中 AGE 大于 20 的所有数据：

```
import mysql.connector

#打开数据库连接
db = mysql.connector.connect(user = 'testuser', password = 'test123', host = '127.0.0.1',
database = 'TESTDB')

#使用 cursor()方法获取操作游标
cursor = db.cursor()

#SQL 删除语句
sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)
try:
    #执行 SQL 语句
    cursor.execute(sql)
    #提交修改
    db.commit()
except:
    #发生错误时回滚
    db.rollback()

#关闭连接
db.close()
```

事务机制可以确保数据的一致性。

事务应该具有原子性、一致性、隔离性、持久性，这 4 个属性通常称为 ACID 特性。

- 原子性(Atomicity)：一个事务是一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做。
- 一致性(Consistency)：事务必须使数据库从一个一致性状态变到另一个一致性状态。一致性与原子性是密切相关的。
- 隔离性(Isolation)：一个事务的执行不能被其他事务干扰，即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。
- 持久性(Durability)：持续性也称永久性(Permanence)，指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的，接下来的其他操作或故障不应该对其有任何影响。

Python DB API 2.0 的事务提供了 commit()和 rollback()两个方法：

```
#SQL 删除记录语句
sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)
try:
    #执行 SQL 语句
    cursor.execute(sql)
    #向数据库提交
    db.commit()
except:
    #发生错误时回滚
    db.rollback()
```

对于支持事务的数据库，在 Python 数据库编程中，当游标建立之时就自动开始了一个隐形的数据库事务。

commit()方法执行所有更新操作，rollback()方法回滚当前游标的所有操作。每一个方法都开始了一个新的事务。

## 9.4 非关系型数据库

### 9.4.1 NoSQL 介绍

NoSQL 指的是非关系型数据库。NoSQL 有时也是 Not Only SQL 的缩写，是对不同于传统关系型数据库的数据库管理系统的统称。

NoSQL 通常用于超大规模数据的存储。这类数据存储不需要固定的模式，无须多余操作就可以横向扩展。现在用户通过第三方平台(例如 Google、Facebook 等)可以很容易地访问和抓取数据。用户的个人信息、社交网络、地理位置、用户生成的数据和用户操作日志已经急剧增加，如果要对这些数据进行存储和挖掘，SQL 数据库已经不适合，NoSQL 数据库却能很好地处理这些大数据。

由于 NoSQL 出现的时间不长，目前对 NoSQL 并没有较为清晰和统一的严谨定义，但是其中的一些主要特征得到了业内人士的公认：其一，NoSQL 数据库不使用结构化查询语言(SQL)，至少没有使用广义上标准的 SQL；其二，所有的 NoSQL 数据库目前都是开源项目，这也是其获得快速发展的最大原因；其三，大多数 NoSQL 数据库都是面向集群或以集群为目的进行开发的，因此更适用于大规模数据存储系统。

与主流关系型数据库采用基于行的存储方式不同，当前的 NoSQL 呈现出多元化

的存储方式发展状况，大体上分为 4 类，即面向文档存储(Document-Oriented)、列存储(Column-Family)、图形关系存储(Graph-Oriented)和键值存储(Key-Value)。

(1) 面向文档存储：面向文档存储将数据以文档的形式储存。每个文档具有自述性，在语义上是自包含的数据单元，拥有分层的树状结构，其格式可以使用可扩展标记语言(XML)、JavaScript 对象标记(JSON)、二进制序列化文档格式(BSON)等。文档由数据项组成，每个数据项都有一个名称及与其对应的值，该值既可以是简单的数据类型，例如纯量值或字符串等，也可以是复杂的数据类型，例如数组集合、对象，甚至可以是另一文档。面向文档存储的最小单位是文档，各文档的结构可以不同，因此是以模式自由(schema-free)方式组织的。

(2) 列存储：列存储将数据储存在列族中，列族中的行把许多列数据与本行的行键关联，一个列族单独存放，保存经常被一起查询的相关数据。由于查询中的选择规则通过列来定义，因此整个数据库是自动索引化的。每个字段的数据聚集存储，在查询少量字段的时候能大大减少读取的数据量。另外，由于列存储是一个字段的数据聚集存储，因此更容易为其设计更好的压缩/解压缩算法。

(3) 图形关系存储：图形关系存储将数据以图的方式储存。实体会被作为结点(Node)，具有属性，而实体之间的关系作为边(Edge)，具有方向性和属性。在用结点和边建立好图之后就可以用多种方式进行查询了。由于在遍历连接和关系时速度很快，所以图形关系存储适合于表达需要强调实体与实体之间关系的数据。

(4) 键值存储：键值存储类似一张哈希表，可以通过主键进行数据的增/删/查/改。该存储方式最大的特点是简单，每个键对应的值仅代表一块数据，无须考虑其结构化信息。

表 9.2 展示了 NoSQL 的主要存储方式及其特点。

**表 9.2　NoSQL 的不同存储方式的特点**

| 存储方式 | 描　述 | 适　用 | 数据量级 | 代表实现 |
|---|---|---|---|---|
| 面向文档存储 | 以文档方式存储非结构化数据 | 非结构化、半结构化数据 | TB～PB | MongoDB |
| 列存储 | 将数据存储在列族中 | 日志 | TB～PB | Cassandra |
| 图形关系存储 | 以结点和边的网络结构存储实体和关系 | 关系性强的数据 | GB～TB | Neo4j |
| 键值存储 | 以哈希表结构存储键值对 | 会话、配置文件、参数 | GB～TB | Redis |

### 9.4.2 MongoDB


视频讲解

MongoDB是一个高性能、高可用性、易于扩展的文档型数据库。在MongoDB中,数据的概念组织形式如下:每个实例包含若干"数据库",每个数据库包含若干"集合"(collection),在集合中又可插入若干条文档。其概念与关系型数据库概念的对应关系如表9.3所示。

**表9.3 MongoDB中各概念与关系型数据库中概念的对应关系**

| MongoDB | 关系型数据库 |
|---|---|
| MongoDB实例 | 数据库实例 |
| 数据库 | 模式 |
| 集合 | 表 |
| 文档 | 行 |
| 数据项 | 字段 |

在MongoDB中,数据是以弹性模式(flexible schema)进行组织的,这也是其与结构化数据库(以表格的行/列结构化形式存储和查询数据)的最大不同之处。在结构化数据库中必须在存储数据之前明确确定表格的模式结构,而MongoDB的集合并不要求事先定义文档的结构。这种弹性模式使得在文档和实体或对象之间进行映射更为方便。

MongoDB以JSON风格的语法格式在文档中表达数据,使用JSON的变种——BSON作为内部存储的格式,针对MongoDB的操作都使用JSON风格的语法,客户端提交或接收的数据也都采用JSON格式的形式来展现:

```
{ "ID": "0001", "name": "ZhangSan", "age": 25 }
```

其中,"ID"、"name"、"age"为文档的键,"0001"、"ZhangSan"、25分别为这3个键所对应的值。可以看到,"age"键所对应的值为数值型对象,其他值为字符串型。此外,文档的值可以是更加复杂的形式,例如数组或内嵌的文档。

与标准SQL不同,在MongoDB中采用find关键字进行数据的查询检索,查询的结果是一个集合中文档的子集,其范围从0个文档到整个集合。MongoDB所支持的find(查询)语句非常强大,其语法类似于面向对象的查询语言,几乎可以实现类似关系数据库单表查询的绝大部分功能,而且支持对数据建立索引。find语句可以以参数表明查询的条件,其形式也是一个JSON文档,若不指定查询条件,默认返回整个

集合的文档：

```
/*返回整个集合的文档*/
db.collection.find();
```

若要查询姓名为"ZhangSan"的用户文档，可以指定 find 语句的首个参数（以 JSON 对象表示）：

```
/*返回姓名为"ZhangSan"的文档*/
db.users.find({"name": "ZhangSan"});
```

在条件参数中通过多个键值对的组合可以完成多条件检索，效果如同 SQL 中的 AND 关键字，例如要查询姓名为"ZhangSan"且性别为"Male"的文档：

```
/*返回姓名为"ZhangSan"且性别为"Male"的文档*/
db.users.find({"name": "ZhangSan", "gender": "Male"});
```

若要求返回指定的键值对而不是整个文档，可通过 find 语句的第二个参数进行指定，该参数同样是 JSON 文档。例如仅需要返回姓名为"ZhangSan"的用户的地址，则：

```
/*返回姓名为"ZhangSan"的用户的地址信息*/
db.users.find({"name": "ZhangSan" }, {"address": 1});
```

在第二个参数中，1 表示需要返回该键值对，0 表示去除结果中的该键值对。

若要求查询的条件在一系列枚举值的范围内，可使用条件操作符 $in。例如需要查询年龄为 25、30 或 35 岁的用户：

```
/*返回年龄为 25、30 或 35 岁的用户信息*/
db.users.find({"age": { $in: [25,30,35]}});
```

若要对查询结果进行排序，可以使用 sort 语句，并可以在其参数中指定要排序的键，其值表明排序的方向，1 表示升序，-1 表示降序，若指定了多个键，则按照指定的顺序逐个排序。例如要对用户以年龄进行升序排序：

```
/*返回所有用户，并对结果按年龄进行升序排序*/
db.users.find().sort({"age": 1});
```

### 9.4.3 PyMongo：MongoDB 和 Python


视频讲解

#### 1. 安装 PyMongo

Python 要连接 MongoDB 需要 MongoDB 驱动，这里使用 PyMongo 驱动来连接。用户可以使用 pip 进行 PyMongo 的安装：

```
$ python -m pip install pymongo
```

用户也可以指定安装的版本：

```
$ python -m pip install pymongo==3.5.1
```

更新 PyMongo 的命令如下：

```
$ python -m pip install --upgrade pymongo
```

接下来可以导入 pymongo 模块，代码如下：

```
import pymongo
```

执行以上代码，如果没有出现错误，表示安装成功。

#### 2. 创建数据库

创建数据库需要使用 MongoClient 对象，并且指定连接的 URL 地址和要创建的数据库名。

例如下例中创建数据库 runoobdb：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["runoobdb"]
```

**注意**：在 MongoDB 中，数据库只有在内容插入之后才会创建。也就是说，在数据库创建后要创建集合(数据表)并插入一个文档(记录)，这样数据库才会真正创建。

可以读取 MongoDB 中的所有数据库，并判断指定的数据库是否存在：

```
import pymongo

myclient = pymongo.MongoClient('mongodb://localhost:27017/')

dblist = myclient.database_names()
if "runoobdb" in dblist:
    print("数据库已存在!")
```

MongoDB 中的集合类似 SQL 中的表。MongoDB 使用数据库对象来创建集合，实例如下：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["runoobdb"]

mycol = mydb["sites"]
```

**注意**：在 MongoDB 中，集合只有在内容插入之后才会创建。也就是说，在创建集合(数据表)后要再插入一个文档(记录)，集合才会真正创建。

可以读取 MongoDB 数据库中的所有集合，并判断指定的集合是否存在：

```
import pymongo

myclient = pymongo.MongoClient('mongodb://localhost:27017/')

mydb = myclient['runoobdb']

collist = mydb.collection_names()
if "sites" in collist:        # 判断 sites 集合是否存在
    print("集合已存在!")
```

MongoDB 中的一个文档类似 SQL 表中的一条记录。

在集合中插入文档使用 insert_one()方法，该方法的第一个参数是字典 name-value 对。

以下实例向 sites 集合中插入文档：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["runoobdb"]
```

```
mycol = mydb["sites"]

mydict = { "name": "RUNOOB", "alexa": "10000", "url": "https://www.runoob.com" }

x = mycol.insert_one(mydict)
print(x)
print(x)
```

insert_one()方法返回 InsertOneResult 对象，该对象包含 inserted_id 属性，它是插入文档的 id 值：

```
import pymongo

myclient = pymongo.MongoClient('mongodb://localhost:27017/')
mydb = myclient['runoobdb']
ycol = mydb["sites"]

mydict = { "name": "Google", "alexa": "1", "url": "https://www.google.com" }

x = mycol.insert_one(mydict)

print(x.inserted_id)
```

用户可以使用 find_one()方法来查询集合中的一条数据。例如查询 sites 文档中的第一条数据：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["runoobdb"]
mycol = mydb["sites"]

x = mycol.find_one()

print(x)
```

find()方法可以查询集合中的所有数据，类似 SQL 中的 SELECT * 操作。

以下实例查找 sites 集合中的所有数据：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["runoobdb"]
mycol = mydb["sites"]
```

```
for x in mycol.find():
    print(x)
```

用户可以在 find()中设置参数来过滤数据。以下实例查找 address 字段为"Park Lane 38"的数据：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["runoobdb"]
mycol = mydb["sites"]

myquery = { "name": "RUNOOB" }

mydoc = mycol.find(myquery)

for x in mydoc:
  print(x)
x = mycol.find_one()

print(x)
```

在查询的条件语句中还可以使用修饰符。以下实例用于读取 name 字段中第一个字母的 ASCII 值大于"H"的数据，大于的修饰符条件为{"$gt"："H"}：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["runoobdb"]
mycol = mydb["sites"]

myquery = { "name": { "$gt": "H" } }

mydoc = mycol.find(myquery)

for x in mydoc:
  print(x)
```

用户可以在 MongoDB 中使用 update_one()方法修改文档中的记录。该方法的第一个参数为查询的条件，第二个参数为要修改的字段。如果查找到的匹配数据超过一条，则只会修改第一条。

以下实例将 alexa 字段的值 10000 改为 12345：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["runoobdb"]
mycol = mydb["sites"]

myquery = { "alexa": "10000" }
newvalues = { "$set": { "alexa": "12345" } }

mycol.update_one(myquery, newvalues)

#输出修改后的 "sites" 集合
for x in mycol.find():
  print(x)
```

update_one()方法只能修改匹配到的第一条记录，如果要修改匹配到的所有记录，可以使用 update_many()。

以下实例将查找所有以 F 开头的 name 字段，并将匹配到的所有记录的 alexa 字段修改为 123：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["runoobdb"]
mycol = mydb["sites"]

myquery = { "name": { "$regex": "^F" } }
newvalues = { "$set": { "alexa": "123" } }

x = mycol.update_many(myquery, newvalues)

print(x.modified_count, "文档已修改")
```

用户可以使用 delete_one()方法来删除一个文档，该方法的第一个参数为查询对象，指定要删除哪些数据。以下实例删除 name 字段值为"Taobao"的文档：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["runoobdb"]
mycol = mydb["sites"]
```

```
myquery = { "name": "Taobao" }

mycol.delete_one(myquery)

# 删除后输出
for x in mycol.find():
  print(x)
```

用户可以使用 delete_many()方法来删除多个文档，该方法的第一个参数为查询对象，指定要删除哪些数据。以下实例删除所有 name 字段中以 F 开头的文档：

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["runoobdb"]
mycol = mydb["sites"]

myquery = { "name": {"$regex": "^F"} }

x = mycol.delete_many(myquery)

print(x.deleted_count, "个文档已删除")
```

## 习题

1. 关系型数据库与非关系型数据库有何异同？

2. 安装 MySQL，并创建数据库 MyDB，使用 Python 编程在该数据库中创建表 Users，它包含 userid、username、password、gender 和 age 5 个字段。

3. 使用 Python 编程在 Users 表中插入如下数据：

| userid | username | password | gender | age |
|---|---|---|---|---|
| 5013 | mikeage | 4303kma | male | 32 |
| 5014 | lineefe | fmeiw12 | female | 48 |
| 5015 | onaverrk | inv8ese | male | 20 |

4. 使用 Python 编程从 Users 表中查询出年龄大于 30 岁的男性用户。

# 第 10 章

# 机器学习——有监督学习

**本章学习目标：**

- 了解机器学习的概念
- 了解有监督和无监督机器学习原理
- 了解有监督机器学习相关算法并进行运用

## 10.1 机器学习简介

机器学习(Machine Learning,ML)是一门多领域交叉学科,涉及概率论、统计学、逼近论、凸分析、算法复杂度理论等多门学科,专门研究计算机怎样模拟或实现人类的学习行为,以获取新的知识或技能,重新组织已有的知识结构使之不断改善自身的性能。它是人工智能的核心,是使计算机具有智能的根本途径。它的应用已遍及人工智能的各个分支,例如专家系统等领域。机器学习有很多学习方法,例如监督学习(supervised learning)、无监督学习(unsupervised learning)、半监督学习(semi-supervised learning)、强化学习(reinforcement learning)等,本章主要介绍有监督学习,本书第 11 章重点介绍无监督学习。

**1. 有监督学习**

有监督学习流程如图 10.1 所示，输入样本(数据)是训练样本，每组训练样本有个明确的标识或结果。在建立预测模型的时候，监督式学习建立一个学习过程，将预测结果与“训练数据”的实际结果进行比较，不断地调整预测模型，直到模型的预测结果达到一个预期的准确率。它常用于回归问题与分类问题。

**2. 无监督学习**

在非监督式学习中，数据并不被特别标识，学习模型是为了推断出数据的一些内在结构。其常见的应用场景包括关联规则的学习以及聚类等。它可以用来解决鸡尾酒会问题，即提取混杂在一起的音频，也可以结合聚类算法来建立图片的 3D 模型等。

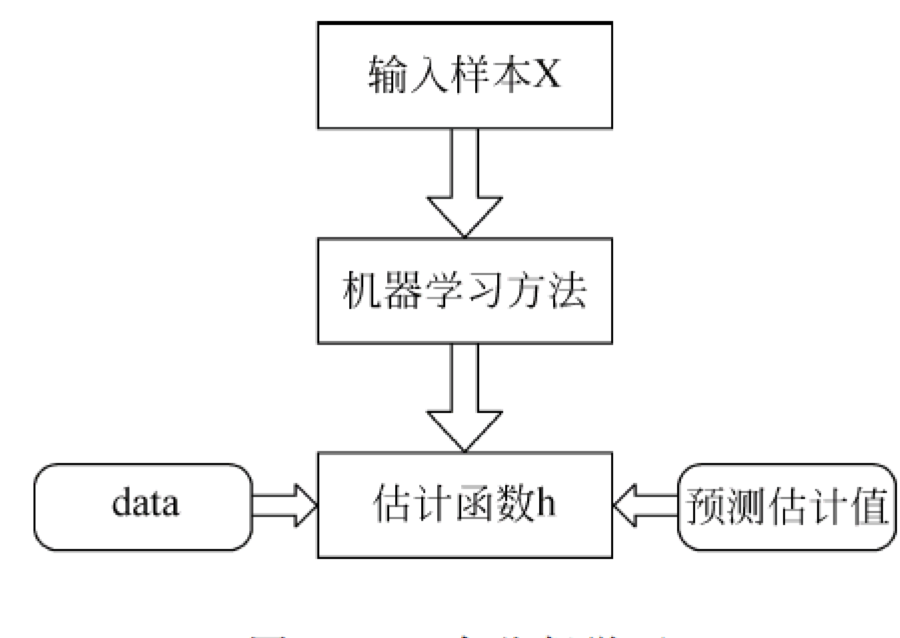



图 10.1 有监督学习

## 10.2 Python 机器学习库 Scikit-learn

Scikit-learn 是一套基于 Python 语言的机器学习库，该库建立在 NumPy、SciPy 和 Matplotlib 基础之上，提供了一整套简单、高效的数据挖掘和数据分析工具。Scikit-learn 发布于 2007 年，已经发展更新了超过 10 年时间，目前已经成为 Python 中最重要、最常用的机器学习工具，集成了大量成熟的机器学习算法。图 10.2 是官方提供的 Scikit-learn 库结构及算法选择流程图。由于 Scikit-learn 模型和算法众多，如何选择合适的模型和算法通常是令用户头疼的事情。该图中是一些大致的指导，圆圈内是判断条件，方框内是可以选择的算法。用户可以根据自己的数据特征和任务目标找到一条合适的操作路线，逐步尝试。

由图 10.2 可见，Scikit-learn 库中主要包含分类、回归、聚类和降维 4 类算法。

分类是指识别给定对象的所属类别，其属于监督学习的范畴，最常见的应用场景包括垃圾邮件检测和图像识别等。目前，Scikit-learn 已经实现的算法包括支持向量机(SVM)、最近邻、逻辑回归、随机森林、决策树以及多层感知器(MLP)神经网络等。

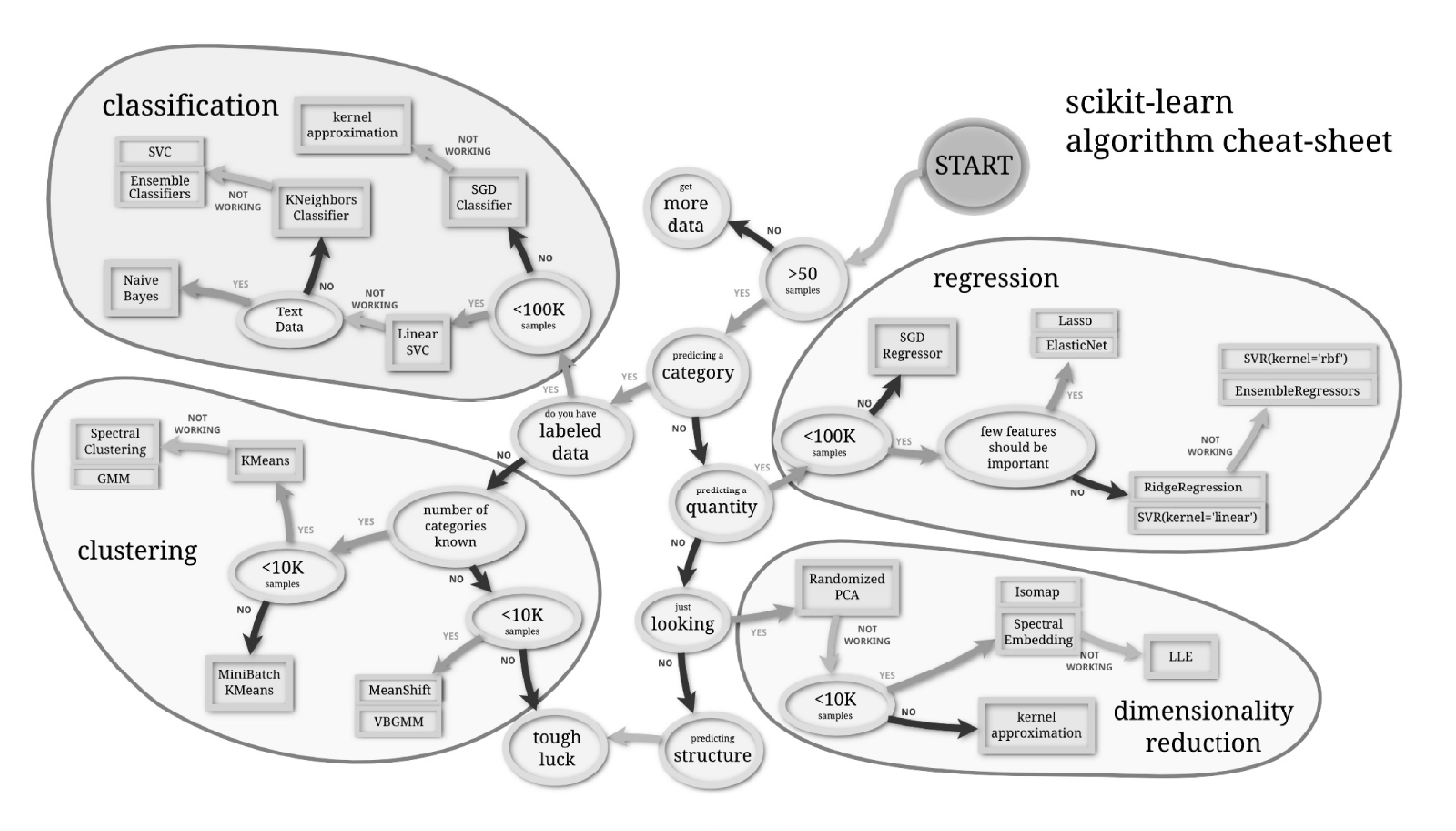



图 10.2 Scikit-learn 库结构及算法选择流程图

需要指出的是，由于 Scikit-learn 本身不支持深度学习，也不支持 GPU 加速，所以这里对于 MLP 的实现并不适合处理大规模问题，有相关需求的读者可以查看同样对 Python 有良好支持的 Keras 和 Theano 等框架。

回归是指预测与给定对象相关联的连续值属性，其最常见的应用场景包括预测药物反应和预测股票价格等。目前，Scikit-learn 已经实现的算法包括支持向量回归(SVR)、脊回归、Lasso 回归、弹性网络(Elastic Net)、最小角回归(LARS )、贝叶斯回归以及各种不同的鲁棒回归算法等。可以看到，这里实现的回归算法几乎涵盖了所有开发者的需求范围，而且更重要的是，Scikit-learn 还针对每种算法提供了简单明了的用例参考。

聚类是指自动识别具有相似属性的给定对象，并将其分组为集合，属于无监督学习的范畴，最常见的应用场景包括顾客细分和试验结果分组。目前，Scikit-learn 已经实现的算法包括 K-Means 聚类、谱聚类、均值偏移、分层聚类、DBSCAN 聚类等。

数据降维是指使用主成分分析(PCA)、非负矩阵分解(NMF)或特征选择等降维技术来减少要考虑的随机变量的个数，其主要应用场景包括可视化处理和效率提升。

此外，Scikit-learn 还具有模型选择和数据预处理的相关功能。

模型选择是指对于给定参数和模型的比较、验证和选择，其主要目的是通过参数调整来提升精度。目前，Scikit-learn 实现的模块包括格点搜索、交叉验证和各种针对预测误差评估的度量函数。

数据预处理是指数据的特征提取和归一化，它是机器学习过程中的第一个也是最重要的一个环节。这里归一化是指将输入数据转换为具有零均值和单位权方差的新变量，但因为大多数时候都做不到精确到零，所以会设置一个可接受的范围，一般要求落在 0～1。特征提取是指将文本或图像数据转换为可用于机器学习的数字变量。

总的来说，Scikit-learn 实现了一整套用于数据降维、模型选择、特征提取和归一化的完整算法/模块，虽然缺少按步骤操作的参考教程，但 Scikit-learn 针对每个算法和模块提供了丰富的参考样例和详细的说明文档。

## 10.3 有监督学习

在机器学习中，有监督学习的任务重点在于根据已有经验知识对未知样本的目标/标记进行预测。根据目标预测变量类型的不同，监督学习任务大体上分为回归预测和分类学习。本章涉及的机器学习算法如图 10.3 所示。

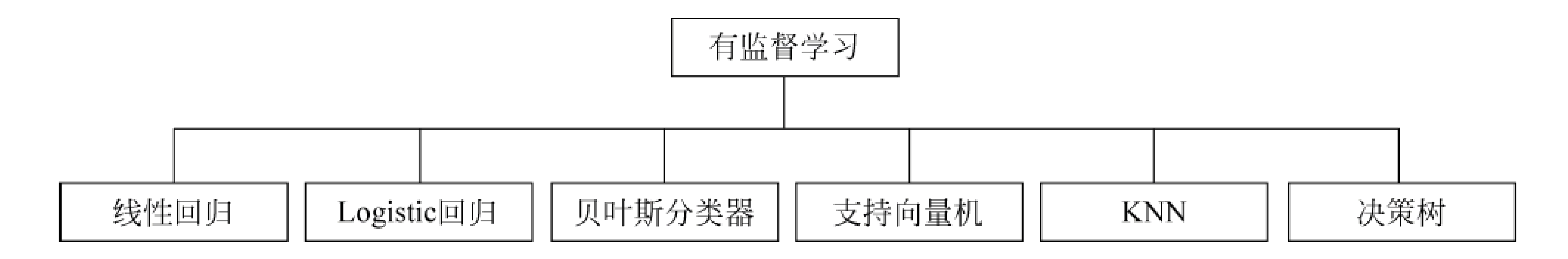



图 10.3　本章涉及的机器学习算法

有监督学习的基本架构和流程如下：首先准备训练数据，这些数据可以是文本数据、图像数据、音频数据等；然后抽取所需要的特征，形成特征向量（Feature Vectors）；接着把这些特征向量以及对应的目标（Labels）一起导入机器学习算法模型中，训练出一个预测模型；然后采用同样的特征抽取方法作用于新数据，得到用于测试的特征向量；最后使用预测模型对这些待测试的特征向量进行预测并得到结果。

### 10.3.1　线性回归


视频讲解

线性回归的目标是提取输入变量与输出变量的关联线性模型，这就要求实际输出与线性方程预测的输出的残差平方和最小化。这种方法也称为最小二乘法。下面用代码进行演示。

（1）导入数据。

```
import numpy as np
X = []
y = []
with open(filename, 'r') as f:
    for line in f.readlines():
        xt, yt = [float(i) for i in line.split(',')]
        X.append(xt)
        y.append(yt)
```

（2）在建立机器学习模型时需要用一种方法来验证模型，检查模型是否达到一定的满意度。为了实现这个方法，把数据分成两组——训练集（training dataset）和测试集（testing dataset）。训练集用来建立模型，测试集用来验证模型对未知数据的学习效果。因此，先把数据分为训练集和测试集。

```
# 切分训练和测试数据
num_training = int(0.8 * len(X))
```

```
num_test = len(X) - num_training

#训练数据
X_train = np.array(X[:num_training]).reshape((num_training,1))
y_train = np.array(y[:num_training])
#测试数据
X_test = np.array(X[num_training:]).reshape((num_test,1))
y_test = np.array(y[num_training:])
```

在这里,把80%的数据作为训练数据集,其余的20%作为测试数据集。

(3) 现在来创建一个线性回归对象。

```
#创建一个线性回归对象
from sklearn import linear_model
linear_regressor = linear_model.LinearRegression()
#用训练数据训练模型
linear_regressor.fit(X_train, y_train)
```

(4) 用训练好的模型对测试集上的数据进行预测。

```
#预测输出结果
y_test_pred = linear_regressor.predict(X_test)
```

(5) 对预测数据进行可视化展示。

```
#可视化展示
import matplotlib.pyplot as plt
plt.scatter(X_test, y_test, color = 'green')
plt.plot(X_test, y_test_pred, color = 'black', linewidth = 4)
plt.xticks(())
plt.yticks(())
plt.show()
```

展示结果如图10.4所示。

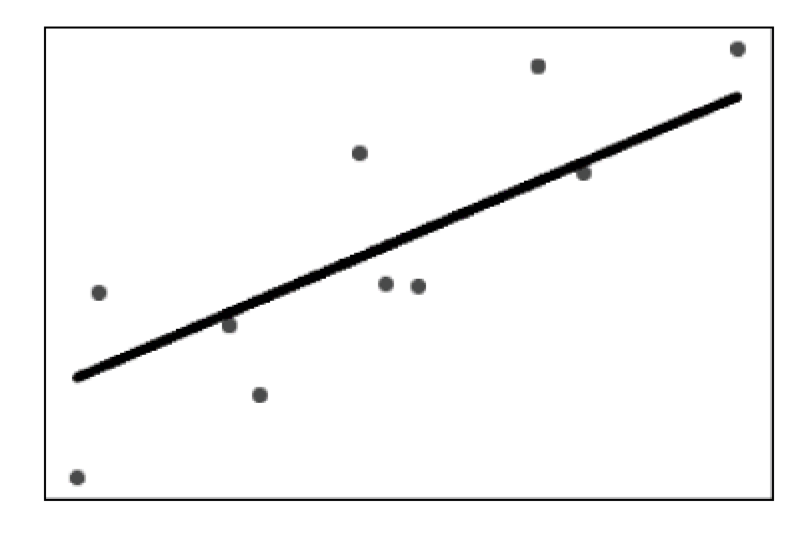


图10.4 代码输出图

(6) 对构建的模型进行评价。

评价一个回归模型的拟合效果主要有以下几个指标。

- **平均绝对误差(mean absoult error)**:这是给定数据集的所有数据点的误差的平方的平均值。

- **均方误差(mean squared error)**：这是给定数据集的所有数据点的误差的平方的平均值。
- **中位数绝对误差(median absoult error)**：这是给定数据集的所有数据点的误差的中位数。这个指标的主要优点是可以消除异常值的干扰。测试数据集中的单个坏点不会影响整个误差指标，均值误差指标会受到异常值的干扰。
- **解释方差得分(explained variance score)**：这个分数用于衡量模型对数据集波动的解释能力。如果得分为1.0，那么表明模型是完美的。
- **R方得分(R2 score)**：这个指标读作"R方"，是指确定性相关系数，用于衡量模型对位置样本预测的效果。其最好的得分是1.0，得分值也可以是负数。

下面用代码来计算这几个指标。

```
import sklearn.metrics as sm

print ("Mean absolute error = ", round(sm.mean_absolute_error(y_test, y_test_pred), 2))
print ("Mean squared error = ", round(sm.mean_squared_error(y_test, y_test_pred), 2))
print ("Median absolute error = ", round(sm.median_absolute_error(y_test, y_test_pred), 2))
print ("Explain variance score = ", round(sm.explained_variance_score(y_test, y_test_
pred), 2))
print ("R2 score = ", round(sm.r2_score(y_test, y_test_pred), 2))
```

得到的结果如下：

```
Mean absolute error = 0.54
Mean squared error = 0.38
Median absolute error = 0.54
Explain variance score = 0.68
R2 score = 0.68
```

以上是一个完整的线性回归模型的建立步骤。最后，对线性回归做一个简单的总结：线性回归器是最简单、易用的回归模型。正因为其对特征和回归目标之间的线性假设，从某种程度上说也局限了其应用范围，特别是现实生活中绝大多数实例数据的特征和目标之间不是严格的线性关系。尽管如此，在不清楚特征之间关系的前提下，仍然可以使用线性回归模型作为大多数科学实验的基线系统(Baseline System)。

### 10.3.2 Logistic 回归分类器

假设现在有一些数据点，使用一条直线对这些点进行拟合(该线称为最佳拟合直线)，这个拟合过程就称作回归。利用 Logistic 回归进行分类的主要思想是根据现有

数据对分类边界线建立回归公式，以此进行分类。“回归”一词源于最佳拟合，表示要找到最佳拟合参数集，其背后的数学分析将在下一部分介绍。训练分类器时的做法就是寻找最佳拟合参数，使用的是最优化算法。我们想要的函数应该是能接受所有的输入，然后预测出类别。例如在两个类的情况下，函数输出0或1，这个函数就是二值型输出分类器的sigmoid函数：

$$g(z)=\frac{1}{1+e^{-z}} \tag{10.1}$$

图10.5给出了sigmoid函数在不同坐标尺度下的两条曲线图。当x为0时，sigmoid函数值为0.5。随着x的增大，对应的sigmoid函数值将逼近于1；随着x的减小，sigmoid函数值将逼近于0。

因此，为了实现Logistic回归分类器，可以在每个特征上都乘以一个回归系数，然后把所有的结果值相加，将这个总和代入sigmoid函数中，进而得到一个范围在0～1的数值。任何大于0.5的数据被归入1类，小于0.5的被归入0类。所以，Logistic回归可以被看成是一种概率估计。

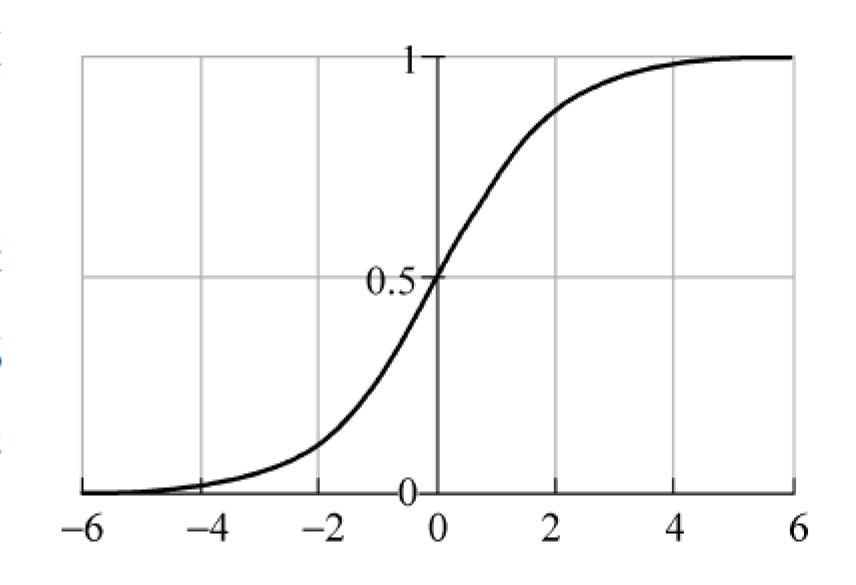


图10.5　Logistic函数图像

在线性回归 $f(x)=wx+b$ 中，为了将整个目标值压缩在(0,1)上，引入Logistic函数，于是就有了

$$g(f(x))=\frac{1}{1+e^{-(wx+b)}} \tag{10.2}$$

Logistic回归的一般过程如下。

(1) 收集数据：采用任意方法收集数据。

(2) 准备数据：由于需要进行距离计算，所以要求数据类型为数值型。另外，结构化数据格式最佳。

(3) 分析数据：采用任意方法对数据进行分析。

(4) 训练算法：大部分时间将用于训练，训练的目的是为了找到最佳的分类回归系数。

(5) 测试算法：一旦训练步骤完成，分类将会很快。

(6) 使用算法：首先需要输入一些数据，并将其转换成对应的结构化数值；接着基于训练好的回归系数就可以对这些数值进行简单的回归计算，判定它们属于哪个类别；在这之后，就可以在输出的类别上做一些具体的分析工作。

下面用 Python 代码来进行良/恶性乳腺肿瘤预测的实践，用到的原始数据下载地址为"https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/"。

(1) 导入需要的数据。

```
# 导入需要用到的库
import pandas as pd
import numpy as np
# 创建特征列表
column_names = ['Sample code number', 'Clump Thickness', 'Unigormity og Cell Size','Uniformity
                of Cell Shape', 'Marginal Adhesion', 'Single Epithelital CellSize', 'Bare
                Nuclei', 'Bland Chromatin', 'Normal Nucleoli', 'Mitoses', 'Class']
# 读取数据
data = pd.read_csv('https://archive.ics.uci.edu/ml/machine - learning - databases/breast -
                   cancer - wisconsin/breast - cancer - wisconsin.data', names = column_
                   names)
data.describe()
```

读取后，数据显示如下。

| | Sample code number | Clump Thickness | Unigormity og Cell Size | Uniformity of Cell Shape | Marginal Adhesion | Single Epithelital CellSize | Bland Chromatin | Normal Nucleoli | Mitoses | Class |
|---|---|---|---|---|---|---|---|---|---|---|
| count | 6.990000e+02 | 699.000000 | 699.000000 | 699.000000 | 699.000000 | 699.000000 | 699.000000 | 699.000000 | 699.000000 | 699.000000 |
| mean | 1.071704e+06 | 4.417740 | 3.134478 | 3.207439 | 2.806867 | 3.216023 | 3.437768 | 2.866953 | 1.589413 | 2.689557 |
| std | 6.170957e+05 | 2.815741 | 3.051459 | 2.971913 | 2.855379 | 2.214300 | 2.438364 | 3.053634 | 1.715078 | 0.951273 |
| min | 6.163400e+04 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 2.000000 |
| 25% | 8.706885e+05 | 2.000000 | 1.000000 | 1.000000 | 1.000000 | 2.000000 | 2.000000 | 1.000000 | 1.000000 | 2.000000 |
| 50% | 1.171710e+06 | 4.000000 | 1.000000 | 1.000000 | 1.000000 | 2.000000 | 3.000000 | 1.000000 | 1.000000 | 2.000000 |
| 75% | 1.238298e+06 | 6.000000 | 5.000000 | 5.000000 | 4.000000 | 4.000000 | 5.000000 | 4.000000 | 1.000000 | 4.000000 |
| max | 1.345435e+07 | 10.000000 | 10.000000 | 10.000000 | 10.000000 | 10.000000 | 10.000000 | 10.000000 | 10.000000 | 4.000000 |

从该表可以知道，共有 699 条数据，各特征的数据描述如上所示。

(2) 对数据进行预处理。

```
# 对数据进行简单预处理
# 将?替换为标准缺失值
data = data.replace(to_replace = '?', value = np.nan)
# 去掉带有缺失值的数据
data = data.dropna(how = 'any')
# 再次对数据进行描述
data.describe()
```

描述结果如下，可以看到去掉缺失值后有683条完整的数据。

| | Sample code number | Clump Thickness | Unigormity og Cell Size | Uniformity of Cell Shape | Marginal Adhesion | Single Epithelital CellSize | Bland Chromatin | Normal Nucleoli | Mitoses | Class |
|---|---|---|---|---|---|---|---|---|---|---|
| count | 6.830000e+02 | 683.000000 | 683.000000 | 683.000000 | 683.000000 | 683.000000 | 683.000000 | 683.000000 | 683.000000 | 683.000000 |
| mean | 1.076720e+06 | 4.442167 | 3.150805 | 3.215227 | 2.830161 | 3.234261 | 3.445095 | 2.869693 | 1.603221 | 2.699854 |
| std | 6.206440e+05 | 2.820761 | 3.065145 | 2.988581 | 2.864562 | 2.223085 | 2.449697 | 3.052666 | 1.732674 | 0.954592 |
| min | 6.337500e+04 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 2.000000 |
| 25% | 8.776170e+05 | 2.000000 | 1.000000 | 1.000000 | 1.000000 | 2.000000 | 2.000000 | 1.000000 | 1.000000 | 2.000000 |
| 50% | 1.171795e+06 | 4.000000 | 1.000000 | 1.000000 | 1.000000 | 2.000000 | 3.000000 | 1.000000 | 1.000000 | 2.000000 |
| 75% | 1.238705e+06 | 6.000000 | 5.000000 | 5.000000 | 4.000000 | 4.000000 | 5.000000 | 4.000000 | 1.000000 | 4.000000 |
| max | 1.345435e+07 | 10.000000 | 10.000000 | 10.000000 | 10.000000 | 10.000000 | 10.000000 | 10.000000 | 10.000000 | 4.000000 |

(3) 由于原始的数据没有提供对应的测试样本用于评估模型，所以需要对标记的数据进行分割，这里用25%的数据作为测试集，75%的数据作为训练集，代码如下。

```
#对数据进行分割
from sklearn.cross_validation import train_test_split
#75%的数据作为训练集,25%的数据作为测试集
X_train, X_test, y_train, y_test = train_test_split(data[column_names[1:10]], data[column
                                   _names[10]], test_size = 0.25, random_state = 33)

#查验训练样本的数量和类别分布
y_train.value_counts()

#查验测试样本的数量和类别分布
y_test.value_counts()
```

以上完成了对数据的预处理过程，用于训练的样本有512条(344条良性肿瘤数据，168条恶性肿瘤数据)，测试样本有171条(100条良性肿瘤数据，71条恶性肿瘤数据)。

(4) 用Logistic回归对上面的数据进行训练。

```
#导入需要用到的算法
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

#对数据进行标准化处理
ss = StandardScaler()
X_train = ss.fit_transform(X_train)
X_test = ss.transform(X_test)

#初始化 Logistic 回归器和 SGDClassifier
lr = LogisticRegression()
```

```
#训练模型
lr.fit(X_train, y_train)

#预测
lr_y_pred = lr.predict(X_test)
```

(5) 模型性能评测。

```
from sklearn.metrics import classification_report
#使用 Logistic 自带的评分函数
print('Accuracy of LR Classifier:', lr.score(X_test, y_test))

#其他指标
print(classification_report(y_test, lr_y_pred, target_names = ['Benign', 'Malignant']))
```

其测评结果如下:

```
Accuracy of LR Classifier: 0.988304093567
             precision    recall  f1 - score   support
  Benign        0.99       0.99      0.99        100
Malignant       0.99       0.99      0.99         71
avg / total     0.99       0.99      0.99        171
```

下面对 Logistic 做一个简单的小结。

(1) 优点：计算代价不高，易于理解和实现。

(2) 缺点：容易欠拟合，分类精度可能不高。

(3) 适用的数据类型：数值型和标称型数据。

## 10.3.3 朴素贝叶斯分类器

朴素贝叶斯是一个非常简单，但实用性很强的分类模型，朴素贝叶斯分类器的构造基础是贝叶斯理论。

抽象一些说，朴素贝叶斯分类器会单独考量每一维度特征被分类的条件概率，进而综合这些概率，并对其所在的特征向量做出分类预测。因此，这个模型的基本数学假设是各个维度上的特征被分类的条件概率之间是相互独立的。

### 1. 贝叶斯决策理论

朴素贝叶斯是贝叶斯决策理论的一部分，所以在讲述朴素贝叶斯之前有必要快

速了解一下贝叶斯决策理论。

假设现在有一个数据集，它由两类数据组成，数据分布如图 10.6 所示。

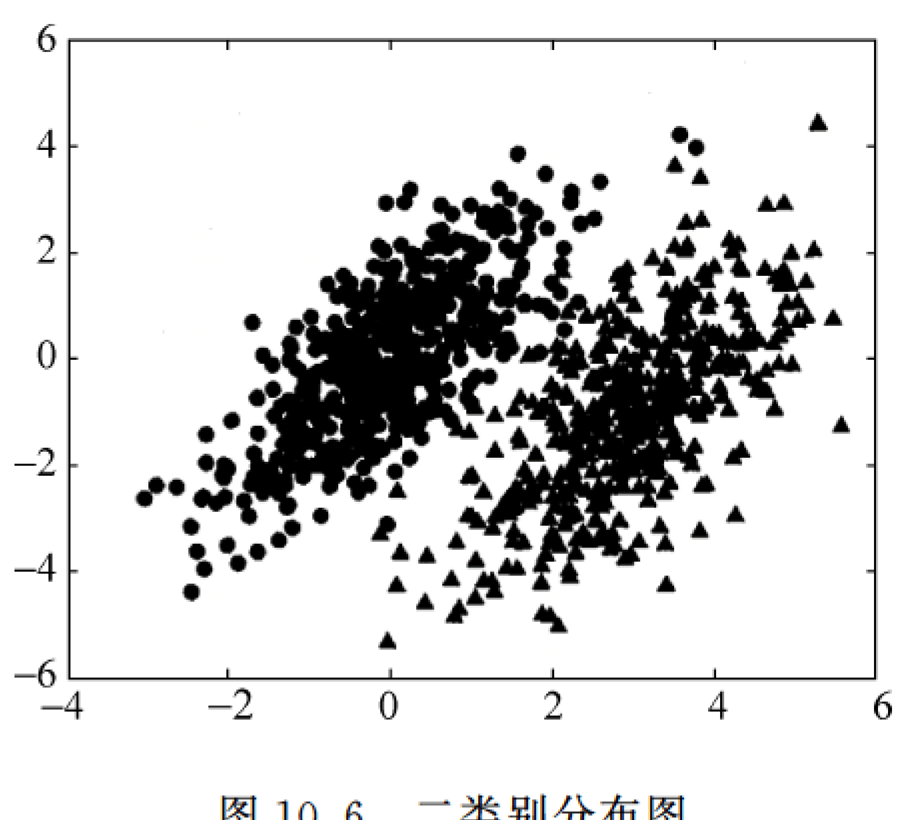


图 10.6 二类别分布图

假设有位读者找到了描述图中两类数据的统计参数。现在用 p1(x,y)表示数据点(x,y)属于类别 1(图中用圆点表示的类别)的概率，用 p2(x,y)表示数据点(x,y)属于类别 2（图中用三角形表示的类别)的概率，那么对于一个新数据点(x,y)，可以用下面的规则来判断它的类别：

(1) 如果 $p1(x,y) > p2(x,y)$，那么类别属于 1。

(2) 如果 $p1(x,y) < p2(x,y)$，那么类别属于 2。

也就是说，我们会选择高概率对应的类别。这就是贝叶斯决策理论的核心思想，即选择具有最高概率的决策。

**2. 朴素贝叶斯(Naïve Bayes)**

根据上面可以知道贝叶斯决策理论是以概率为基础的，那么朴素贝叶斯的思想也很简单，是基于条件概率的：对于给出的待分类项，求解在此项出现的条件下各个类别出现的概率，哪个概率值大就认为该分类项属于哪一类。其定理定义如下：

(1) 设 $x=\{a_1,a_2,\cdots,a_n\}$为待分类项，而每个 $a_i$ 为输入 x 的一个特征属性。

(2) 设 $Y=\{y_1,y_2,\cdots,y_m\}$为一个类别集合。

(3) 计算 $P(y_1|x),P(y_2|x),\cdots,P(y_m|x)$。

(4) 如果 $P(y_k|x)=\max\{P(y_1|x),P(y_2|x),\cdots,P(y_m|x)\}$，则 $x\subset y_k$。

上面定义的关键步骤还是步骤 3，该步的求解就用到了朴素贝叶斯的两大基

础——贝叶斯公式和特征条件独立假设。具体求解过程如下：

(1) 给定一组训练数据集，用于训练参数。

(2) 统计得到在每种类别下各个特征属性的条件概率估计(这一步使用极大似然估计或者贝叶斯估计)。

$$\begin{aligned}&P(a_1 \mid y_1),P(a_2 \mid y_1),\cdots,P(a_n \mid y_1)\\&P(a_1 \mid y_2),P(a_2 \mid y_2),\cdots,P(a_n \mid y_2)\\&\cdots\\&P(a_1 \mid y_m),P(a_2 \mid y_m),\cdots,P(a_n \mid y_m)\end{aligned} \tag{10.3}$$

(3) 根据贝叶斯公式有以下推导：

$$P(y_i \mid x) = \frac{P(x \mid y_i)P(y_i)}{P(x)} \tag{10.4}$$

由全概率公式可知，对于所有类别来说，$P(x)$为一个常数。因此，只需要比较每一类的 $P(x|y_i)P(y_i)$，哪个值最大，待分类项就是哪一类。因为有特征条件独立的假设，所以可以使用条件独立公式求解：

$$P(x \mid y_i)P(y_i) = P(a_1 \mid y_i) * P(a_2 \mid y_i) * \cdots * P(a_n \mid y_i) * P(y_i) \tag{10.5}$$

### 3. 用朴素贝叶斯对文档进行分类

朴素贝叶斯分类器的一个重要应用就是文档的自动分类。在文档分类中，整个文档(例如一封电子邮件)是实例，而电子邮件中的某些元素则构成特征。虽然电子邮件是一种会不断增加的文本，但我们同样可以对新闻报道、用户留言、政府公文等其他任意类型的文本进行分类。可以观察文档中出现的词，并把每个词的出现或者不出现作为一个特征，这样得到的特征数目就会跟词汇表中的词目一样多。朴素贝叶斯是朴素贝叶斯分类器的一个扩展，是用于文档分类的常用算法。

朴素贝叶斯的一般使用过程如下。

(1) 收集数据：可以使用任何方法。

(2) 准备数据：需要数值型或者布尔型数据。

(3) 分析数据：当有大量特征时，绘制特征作用不大，此时使用直方图效果更好。

(4) 训练算法：计算不同的独立特征的条件概率。

(5) 测试算法：计算错误率。

（6）使用算法：一个常见的朴素贝叶斯应用是文档分类。用户可以在任意的分类场景中使用朴素贝叶斯分类器，不一定非要是文本。

下面用 20 类新闻文本数据为例，对其用朴素贝叶斯分类器进行分类。

（1）导入数据。

```
#导入数据
from sklearn.datasets import fetch_20newsgroups
news = fetch_20newsgroups(subset = 'all')
```

（2）对数据做随机分割，形成训练集和测试集。

```
#数据分割
from sklearn.cross_validation import train_test_split
#随机采样,25%的数据样本作为测试集
X_train, X_test, y_train, y_test = train_test_split(news.data, news.target, test_size = 0.25,
                                                    random_state = 33)
```

（3）构建模型。

```
#使用朴素贝叶斯分类器对新闻文本数据进行类别预测
#将文本数据转化为特征向量
from sklearn.feature_extraction.text import CountVectorizer
vec = CountVectorizer()
X_train = vec.fit_transform(X_train)
X_test = vec.transform(X_test)
#导入贝叶斯模型
from sklearn.naive_bayes import MultinomialNB
mnb = MultinomialNB()
#训练模型
mnb.fit(X_train, y_train)
#预测
y_pred = mnb.predict(X_test)
```

（4）模型评估。

```
#性能评估
from sklearn.metrics import classification_report
print('The accuracy of Naive Bayes Classifier is:', mnb.score(X_test, y_test))
print(classification_report(y_test, y_pred, target_names = news.target_names))
```

输出结果如下：

```
The accuracy of Naive Bayes Classifier is: 0.839770797963
                          precision    recall  f1-score   support
             alt.atheism       0.86      0.86      0.86       201
           comp.graphics       0.59      0.86      0.70       250
 comp.os.ms-windows.misc       0.89      0.10      0.17       248
comp.sys.ibm.pc.hardware       0.60      0.88      0.72       240
   comp.sys.mac.hardware       0.93      0.78      0.85       242
          comp.windows.x       0.82      0.84      0.83       263
            misc.forsale       0.91      0.70      0.79       257
               rec.autos       0.89      0.89      0.89       238
         rec.motorcycles       0.98      0.92      0.95       276
      rec.sport.baseball       0.98      0.91      0.95       251
        rec.sport.hockey       0.93      0.99      0.96       233
               sci.crypt       0.86      0.98      0.91       238
         sci.electronics       0.85      0.88      0.86       249
                 sci.med       0.92      0.94      0.93       245
               sci.space       0.89      0.96      0.92       221
  soc.religion.christian       0.78      0.96      0.86       232
      talk.politics.guns       0.88      0.96      0.92       251
   talk.politics.mideast       0.90      0.98      0.94       231
      talk.politics.misc       0.79      0.89      0.84       188
      talk.religion.misc       0.93      0.44      0.60       158

             avg / total       0.86      0.84      0.82      4712
```

通过代码的输出可以知道，朴素贝叶斯对该文本数据分类的正确率为83.977%，平均精确率、召回率和F1得分分别为0.86、0.84和0.82。

朴素贝叶斯模型被广泛应用于海量互联网文本的分类任务。由于其较强的特征条件独立假设，使得模型预测所需要估计的参数规模从幂指数量级向线性量级减少，极大地节约了内存消耗和计算时间。但是，也正是因为受到这种强假设的限制，模型训练时无法将各个特征之间的联系考虑进去，使得该模型在其他数据特征关联性较强的分类任务上的性能表现不佳。

小结：对于分类而言，使用概率有时要比使用硬规则更为有效。贝叶斯概率及贝叶斯准则提供了一种利用已知值来估计未知概率的有效方法，可以通过特征之间的条件独立性假设降低对数据量的需求。独立性假设是指一个词的出现概率并不依赖于文档中的其他词。当然，这个假设过于简单。这就是称之为朴素贝叶斯的原因。尽管条件独立性假设并不正确，但是朴素贝叶斯仍然是一种有效的分类器。

### 10.3.4　支持向量机

在介绍支持向量机(SVM)之前先解释几个概念。如图 10.7 所示,将数据集分隔开来的直线称为分隔超平面(separating hyper plane)。在下面给出的例子中,由于数据点都在二维平面上,所以此时分隔超平面只是一条直线。

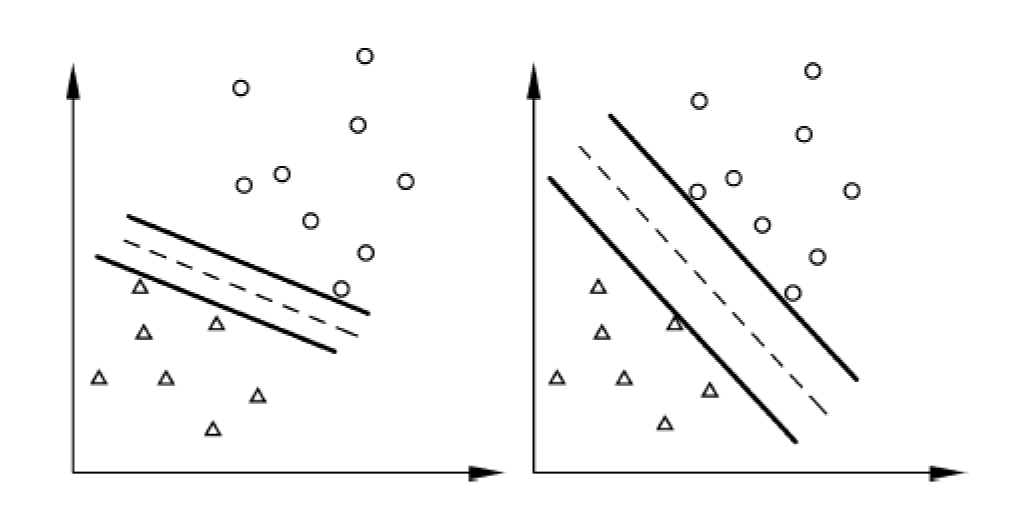


图 10.7　分隔超平面

如图 10.7 所示,从直观上来看,左边肯定不是最优分界面;而右边能让人感觉到其距离更大,使用的支撑点更多,应该是最优分界面。那么,究竟什么样的分界面是最优的呢?

采用这种方式来构建分类器,如果数据点离决策边界越远,那么其最后的预测结果也就越可信。所以我们希望找到离分隔超平面最近的点,确保它们离分隔面的距离尽可能远。这里点到分隔面的距离被称为间隔(margin)。希望间隔尽可能大,这是因为如果我们犯错或者在有限数据上训练分类器,我们希望分类器尽可能健壮。

支持向量(support vector)就是离分隔超平面最近的那些点。接下来要试着最大化支持向量到分隔面的距离,需要找到此问题的优化求解方法。

支持向量机分类器是根据训练样本的分布搜索所有可能的线性分类器中最佳的那个,即寻找到一个最佳超平面。

接下来用 Scikit-learn 内部集成的手写体数字图片的数据集对 SVM 进行应用。

```
#读取数据
from sklearn.datasets import load_digits
digits = load_digits()
#读取数据维度
digits.data.shape
(1797, 64)
```

从上面这段代码的运行结果可以知道该手写体数字的数码图像数据共有 1797 条，并且每幅图片都是由 8×8＝64 的像素矩阵组成。在模型使用这些像素矩阵的时候，我们习惯将 2D 的图片像素矩阵逐行首尾拼接成 1D 的像素特征向量。这样做也许会损失一些数据本身的结构信息，但遗憾的是，我们当下所介绍的经典模型都没有对结构性信息进行学习的能力。

按照惯例，对于没有直接提供测试样本的数据，我们都要通过数据分割获取 75％的训练样本和 25％的测试样本数据，代码如下：

```
#对数据进行分割
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(digits.data, digits.target, test_size
= 0.25, random_state = 33)
```

接下来用上面产生的训练数据来训练一个基于线性核函数的支持向量机模型：

```
#使用 SVM 对数字进行识别
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC
#初始化
ss = StandardScaler()
#数据标准化
X_train = ss.fit_transform(X_train)
X_test = ss.transform(X_test)

#训练模型
svc = LinearSVC()
svc.fit(X_train, y_train)
#预测
y_pred = svc.predict(X_test)
#模型评估
print ('The accuracy of SVM is:', svc.score(X_test, y_test))
from sklearn.metrics import classification_report
```

代码输出结果为：

```
The accuracy of SVM is: 0.953333333333
             precision    recall  f1 - score   support
          0       0.92      1.00       0.96        35
          1       0.96      0.98       0.97        54
          2       0.98      1.00       0.99        44
          3       0.93      0.93       0.93        46
          4       0.97      1.00       0.99        35
```

```
             5       0.94       0.94       0.94         48
             6       0.96       0.98       0.97         51
             7       0.92       1.00       0.96         35
             8       0.98       0.84       0.91         58
             9       0.95       0.91       0.93         44
avg / total        0.95       0.95       0.95       450
```

通过代码的运行结果可以知道，支持向量机模型的确能够提供比较高的手写数字识别性能。平均而言，各项指标都在95%左右。

在这里需要指出的是，召回率、准确率和F1得分指标最先适用于二分类任务，但是在本例中分类目标有10个，即数字0～9，因此无法直接计算上述3个指标。通常的做法是逐一评估某个类别的这3项指标：把所有其他的类别看作是负样本，这样一来，就创造了10个二分类问题。

SVM的特点分析：支持向量机模型曾经在机器学习研究领域繁荣发展了很长一段时间，主要是在于其精妙的模型设计，它可以帮助用户在海量甚至高维度的数据中筛选对预测任务最为有效的少数训练样本。这样做不仅节省了学习所需要的数据内存，同时也提高了模型的预测性能。然而，要获得如此的优势就必须付出更多的计算代价(CPU资源和计算时间)。

最后对SVM做一个简单的总结，包括它的一些显著特点以及一些缺点。

(1) 支持向量的重要性：SVM的计算复杂度主要取决于支持向量的个数，而不是样本空间的维数，所以在一定程度上避免了维数灾难，这是很难的。另外，支持向量具有鲁棒性，因为SVM的分类主要是由支持向量来确定的，所以对于那些不是支持向量的样本数据，并不会对最后的分类影响多少。

(2) 核函数的威力：虽然SVM是线性的学习器，但是它借助核函数可以实现低位非线性向量到高位空间线性向量的映射，这使得SVM不仅可以解决线性问题，也可以解决非线性问题。

(3) SVM是使用二次规划进行求解的，所以在求解过程中需要的存储空间比较大，这导致SVM适合小样本的数据，而对于规模比较大的数据效果不会很好，这也是SVM的重要缺点。

### 10.3.5　KNN算法



视频讲解

K邻近算法，或者说K最近邻(k-Nearest Neighbor，KNN)分类算

法，是数据挖掘分类技术中最简单的方法之一。所谓 K 最近邻，是 K 个最近的邻居的意思，即每个样本都可以用它最接近的 K 个邻居来代表。

### 1. KNN 算法的思想

如果一个样本在特征空间中的 K 个最相邻的样本中的大多数属于某一个类别，则该样本也属于这个类别，并具有这个类别上样本的特性。该方法在确定分类决策上只依据最邻近的一个或者几个样本的类别来决定待分样本所属的类别。KNN 方法在用于类别决策时只与极少量的相邻样本有关。

由于 KNN 方法主要靠周围有限的邻近样本，而不是靠判别类域的方法来确定所属的类别，所以对于类域的交叉或重叠较多的待分样本集来说，KNN 方法比其他方法更为适合。

### 2. KNN 算法的决策过程

图 10.8 中有两种类型的样本数据，一类是正方形，另一类是三角形，中间圆形是待分类数据。

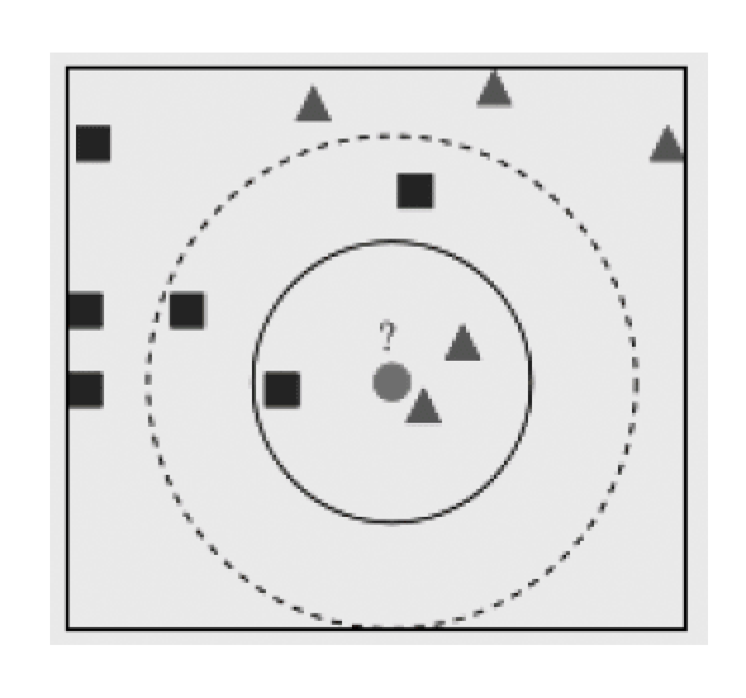


图 10.8 K 近邻分类图

如果 K=3，那么离圆点最近的有两个三角形和一个正方形，这 3 个点进行投票，于是待分类点就属于三角形。如果 K=5，那么离圆点最近的有两个三角形和 3 个正方形，这 5 个点进行投票，于是待分类点就属于正方形。

KNN 算法不仅可以用于分类，还可以用于回归。通过找出一个样本的 K 个最近邻居，将这些邻居的属性的平均值赋给该样本，就可以得到该样本的属性。更有用的方法是将不同距离的邻居对该样本产生的影响给予不同的权值(weight)，例如权值与距离成反比。

### 3. KNN 算法的 Python 实现

下面用代码来实现 KNN 算法的应用。本次用到的数据是经典的 Iris 数据集。该数据集有 150 条鸢尾花数据样本，并且均匀地分布在 3 个不同的亚种，每个数据样本被 4 个不同的花瓣、花萼的形状特征所描述。

```
#读取数据
from sklearn.datasets import load_iris
data = load_iris()
#查看数据大小
data.data.shape
(150, 4)
#查看数据说明
print (data.DESCR)
Notes
-----
Data Set Characteristics:
    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris - Setosa
                - Iris - Versicolour
                - Iris - Virginica
    :Summary Statistics:

    ============== ==== ==== ======= ===== ====================
                    Min  Max   Mean    SD   Class Correlation
    ============== ==== ==== ======= ===== ====================
    sepal length:   4.3  7.9   5.84   0.83    0.7826
    sepal width:    2.0  4.4   3.05   0.43   -0.4194
    petal length:   1.0  6.9   3.76   1.76    0.9490  (high!)
    petal width:    0.1  2.5   1.20   0.76    0.9565  (high!)
    ============== ==== ==== ======= ===== ====================

    :Missing Attribute Values: None
    :Class Distribution: 33.3% for each of 3 classes.
    :Creator: R.A. Fisher
    :Donor: Michael Marshall (MARSHALL%PLU@io.arc.nasa.gov)
    :Date: July, 1988
This is a copy of UCI ML iris datasets.
http://archive.ics.uci.edu/ml/datasets/Iris
The famous Iris database, first used by Sir R.A Fisher
This is perhaps the best known database to be found in the pattern recognition literature.
Fisher's paper is a classic in the field and is referenced frequently to this day.  (See Duda
& Hart, for example.)  The data set contains 3 classes of 50 instances each, where each class
refers to a type of iris plant.  One class is linearly separable from the other 2; the latter
are NOT linearly separable from each other.
```

```
References
----------
   - Fisher,R.A. "The use of multiple measurements in taxonomic problems"
     Annual Eugenics, 7, Part II, 179-188 (1936); also in "Contributions to
     Mathematical Statistics" (John Wiley, NY, 1950).
   - Duda,R.O., & Hart,P.E. (1973) Pattern Classification and Scene Analysis.
     (Q327.D83) John Wiley & Sons.  ISBN 0-471-22361-1.  See page 218.
   - Dasarathy, B.V. (1980) "Nosing Around the Neighborhood: A New System
     Structure and Classification Rule for Recognition in Partially Exposed
     Environments".  IEEE Transactions on Pattern Analysis and Machine
     Intelligence, Vol. PAMI-2, No. 1, 67-71.
   - Gates, G.W. (1972) "The Reduced Nearest Neighbor Rule".  IEEE Transactions
     on Information Theory, May 1972, 431-433.
   - See also: 1988 MLC Proceedings, 54-64.  Cheeseman et al"s AUTOCLASS II
     conceptual clustering system finds 3 classes in the data.
   - Many, many more ...
```

通过上述代码对数据的查验以及数据本身的描述，我们可以了解到 Iris 数据集共有 150 条鸢尾花数据样本，并且均匀地分布在 3 个不同的亚种，每个数据样本被 4 个不同的花瓣、花萼的形状特征所描述。由于没有指定的测试集，依据惯例，需要对数据进行随机分割，将 25％的数据用作测试，75％的数据用作训练。

需要强调的是，如果读者自行编写程序用作数据分割，请务必保证是随机采样。尽管很多数据集中的样本的排序相对随机，但是也有例外。在本例中，Iris 数据就是根据类别依次排列的。如果只采样前 25％的数据用作测试，那么所有的测试样本都属于一个类别。同时训练样本也是不均衡的，这样得到的结果存在偏置，并且可信度非常低，Scikit-learn 所提供的数据分割模块是默认采用随机采样功能的，因此读者不必担心。

```
#对数据进行分割
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(data.data, data.target, test_size = 0.25,
                                                    random_state = 33)

#使用 KNN 算法进行分类
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
#初始化
ss = StandardScaler()
```

```
#数据标准化
X_train = ss.fit_transform(X_train)
X_test = ss.transform(X_test)

#训练模型
knc = KNeighborsClassifier()
knc.fit(X_train, y_train)
#预测
y_pred = knc.predict(X_test)

#模型评估
print ('The accuracy of KNN is:', knc.score(X_test, y_test))
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred, target_names = data.target_names))
```

代码输出结果如下，KNN 算法对鸢尾花测试数据的分类准确率为 89.474%，其他数据如下。

```
The accuracy of KNN is: 0.894736842105
              precision    recall  f1 - score   support

     setosa       1.00      1.00      1.00         8
versicolor        0.73      1.00      0.85        11
  virginica       1.00      0.79      0.88        19

avg / total       0.92      0.89      0.90        38
```

KNN 算法的特点分析：KNN 算法是非常直观的机器学习模型，因此深受广大初学者的喜爱。许多教科书往往以此模型抛砖引玉，足以看出其不仅特别，而且尚有瑕疵之处。细心的读者会发现，KNN 算法与其他算法模型最大的不同在于该模型没有参数训练过程。也就是说，我们并没有通过任何学习算法来分析训练数据，而只是根据测试样本在训练数据中的分布直接做出分类决策。因此，KNN 算法属于无参数模型中非常简单的一种。然而，正是这样的决策算法导致了其非常高的计算复杂度和内存消耗。因为该模型每处理一个测试样本，都需要对所有事先加载在内存中的训练样本进行遍历、逐一计算相似度、排序并且选取 K 个最近邻训练样本的标记，进而做出分类决策。这是平方级的算法复杂度，一旦数据规模稍大，便需要权衡更多计算时间的代价。

最后，对 KNN 算法做一个简单的小结。

优点：

(1) 简单，易于理解，易于实现，无须估计参数，无须训练。

(2) 适合对稀有事件进行分类。

(3) 特别适合于多分类问题(multi-modal，对象具有多个类别标签)，KNN 比 SVM 的表现要好。

缺点：

(1) 当样本不平衡时，例如一个类的样本容量很大，而其他类的样本容量很小，有可能导致输入一个新样本时该样本的 K 个邻居中大容量类的样本占多数，少数类容易分错。

(2) 需要存储全部训练样本。

(3) 计算量较大，因为对每一个待分类的文本都要计算它到全体已知样本的距离，才能求得它的 K 个最近邻点。

(4) 可理解性差，无法给出像决策树那样的规则。

### 10.3.6 决策树



视频讲解

#### 1. 决策树简介

决策树的原理非常简单，即便用户之前没有接触过决策树，也可以通过图 10.9 了解其工作原理。图 10.9 所示的流程图就是一个决策树，正方形代表判断模块

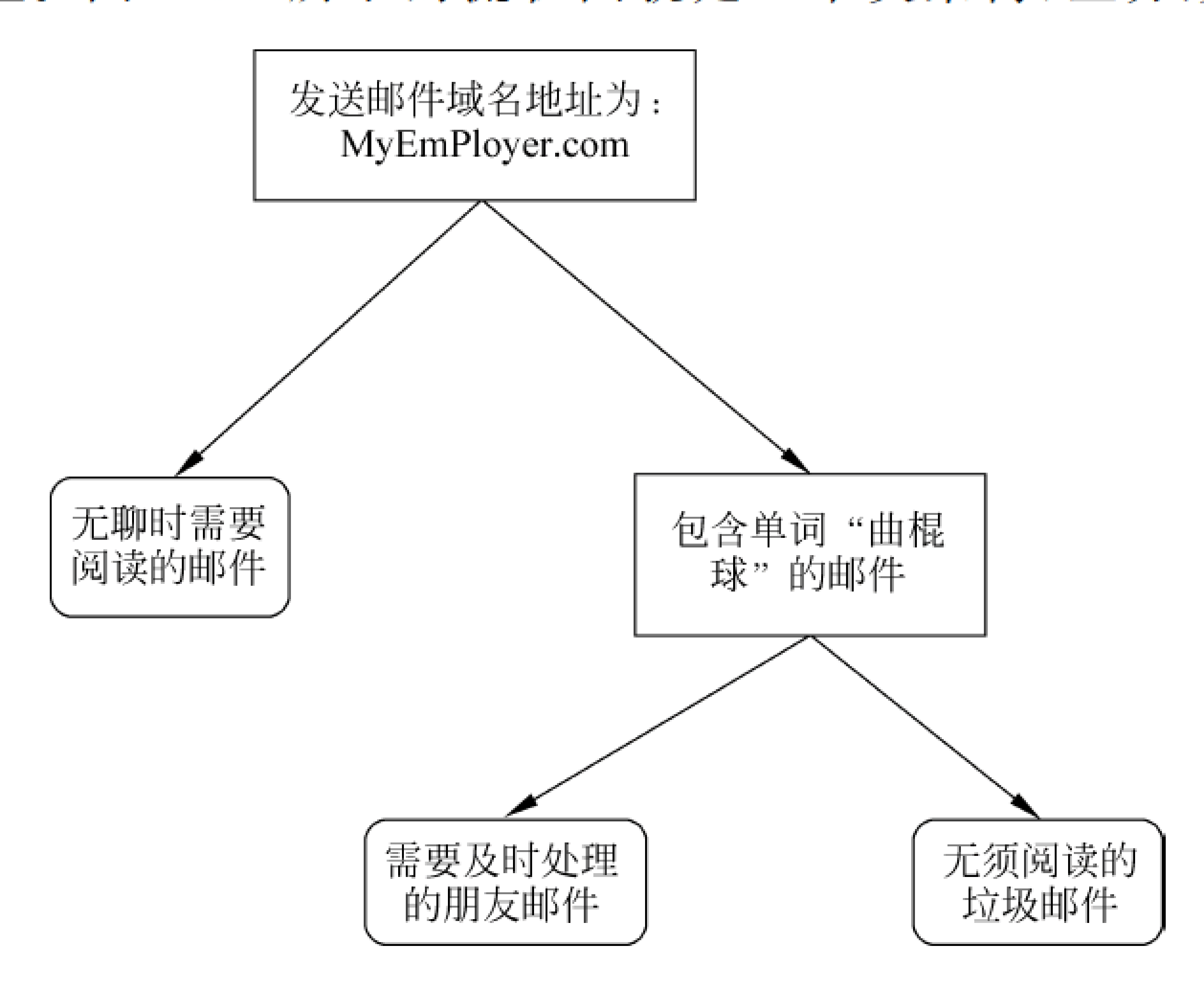



图 10.9 流程图形式的决策树

(decision block),椭圆形代表终止模块(terminating block),表示已经得出结论,可以终止运行。从判断模块引出的左右箭头称作分支(branch),它可以到达另一个判断模块或者终止模块。图 10.9 构造了一个假想的邮件分类系统,它首先检测发送邮件域名地址,如果地址为 MyEmPloyer. com,则将其放在分类“无聊时需要阅读的邮件”中。如果邮件不是来自这个域名,则检查邮件内容里是否包含单词“曲棍球”,若包含,则将邮件归类到“需要及时处理的朋友邮件”;若不包含,则将邮件归类到“无须阅读的垃圾邮件”。

在构造决策树时,需要解决的第一个问题就是当前数据集上哪个特征在划分数据分类时起决定性作用。为了找到决定性的特征,划分出最好的结果,必须评估每个特征。在完成测试之后,原始数据集就被划分为几个数据子集。这些数据子集会分布在第一个决策点的所有分支上 。如果某个分支下的数据属于同一类型,则当前无须阅读的垃圾邮件已经正确地划分数据分类,无须进一步对数据集进行分割。如果数据子集内的数据不属于同一类型,则需要重复划分数据子集的过程。划分数据子集的方法和划分原始数据集的方法相同,直到所有具有相同类型的数据均在一个数据子集内。

目前常用的决策树算法有 ID3 算法、C4. 5 算法和 CART 算法,这些算法将在后面介绍。

**2. 信息增益**

划分数据集的大原则是将无序的数据变得更加有序。用户可以使用多种方法划分数据集,但是每种方法都有各自的优缺点。组织杂乱无章的数据的一种方法就是使用信息论度量信息,信息论是量化处理信息的分支科学。用户可以在划分数据之前使用信息论量化度量信息的内容。

在划分数据集之前、之后信息发生的变化称为信息增益,知道如何计算信息增益,就可以计算每个特征值划分数据集获得的信息增益,获得信息增益最高的特征就是最好的选择。

在可以评测哪种数据划分方式是最好的数据划分之前,必须学习如何计算信息增益。集合信息的度量方式称为香农熵或者简称为熵,这个名字来源于信息论之父克劳德·香农。

熵定义为信息的期望值,在明晰这个概念之前,大家必须知道信息的定义。如果

待分类的事务可能划分在多个分类之中，则信息定义为：

$$l(x_i) = -\log_2 p(x_i) \tag{10.6}$$

其中，$p(x_i)$为选择该分类的概率。

为了计算熵，需要计算所有类别中所有可能值包含的信息期望值，通过下面的公式得到：

$$H = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i) \tag{10.7}$$

其中，n 是分类的数目。

已经有了熵作为衡量训练样本集合的标准，现在可以定义属性分类训练数据的效力的度量标准。这个标准被称为“信息增益(information gain)”。简单地说，一个属性的信息增益就是由于使用这个属性分割样例而导致的期望熵降低(或者说样本按照某属性划分时造成熵减少的期望)。现在假设将训练元组 D 按属性 A 进行划分，则 A 对 D 划分的期望信息为：

$$I_A(D) = -\sum_{j=1}^{v} \frac{D_i}{D} I(D_j) \tag{10.8}$$

信息增益即为两者的差值：

$$\text{gain}(A) = I(A) - I_A(D) \tag{10.9}$$

### 3. 决策树分类算法

1) ID3 算法

ID3 算法是决策树算法的一种。在了解什么是 ID3 算法之前，读者要先明白一个概念——奥卡姆剃刀。

奥卡姆剃刀(Occam's Razor，Ockham's Razor)又称“奥坎的剃刀”，是由 14 世纪逻辑学家、圣方济各会修士奥卡姆的威廉(William of Occam，约 1285 年至 1349 年)提出的，他在《箴言书注》2 卷 15 题说“切勿浪费较多东西，去做‘用较少的东西，同样可以做好的事情’。”简单点说便是 be simple。

ID3 算法是一个由 Ross Quinlan 发明的用于决策树的算法。这个算法便是建立在上述所介绍的奥卡姆剃刀的基础上：越是小型的决策树越优于大的决策树(be simple 理论)。尽管如此，该算法也不是总是生成最小的树形结构，而是一个启发式算法。

从信息论知识中可以知道，期望信息越小，信息增益越大，从而纯度越高。ID3

算法的核心思想就是以信息增益度量属性选择，选择分裂后信息增益最大的属性进行分裂。该算法采用自顶向下的贪婪搜索遍历可能的决策树空间。

所以，ID3 的思想如下：

(1) 自顶向下的贪婪搜索遍历可能的决策树空间构造决策树（此方法是 ID3 算法和 C4.5 算法的基础）。

(2) 从“哪一个属性将在树的根结点被测试”开始。

(3) 使用统计测试来确定每一个实例属性单独分类训练样例的能力，分类能力最好的属性作为树的根结点测试（如何定义或者评判一个属性是分类能力最好的呢？这便是前面介绍的信息增益，或信息增益率。这里要说的是信息增益和信息增益率是不同的，ID3 基于信息增益来选择最好的属性，而接下来介绍的 C4.5 则是基于增益率来进行选择，这也是它进步的地方）。

(4) 然后为根结点属性的每个可能值产生一个分支，并把训练样例排列到适当的分支（也就是说，样例的该属性值对应的分支）之下。

(5) 重复这个过程，用每个分支结点关联的训练样例来选取在该点被测试的最佳属性。

这形成了对合格决策树的贪婪搜索，也就是算法从不回溯重新考虑以前的选择。

2) C4.5 算法

C4.5 是机器学习算法中的另一个分类决策树算法，它是决策树（决策树也就是做决策的结点间的组织方式像一棵树，其实是一个倒树）核心算法，也是前面所介绍的 ID3 的改进算法，所以用户基本上了解了一般决策树构造方法就能构造它。决策树构造方法其实就是每次选择一个好的特征以及分裂点作为当前结点的分类条件。既然说 C4.5 算法是 ID3 的改进算法，那么 C4.5 与 ID3 相比改进的地方有哪些呢？

(1) 用信息增益率来选择属性：ID3 选择属性用的是子树的信息增益，这里可以用很多方法来定义信息，ID3 使用的是熵（entropy，熵是一种不纯度度量准则），也就是熵变化值，而 C4.5 用的是信息增益率。它们的区别就在于一个是信息增益，一个是信息增益率。

(2) 在树的构造过程中进行剪枝，在构造决策树的时候那些挂着几个元素的结点最好不考虑，否则容易导致 over fitting。

(3) 对非离散数据也能处理。

(4) 能够对不完整数据进行处理。

针对上述第一点解释一下：一般来说，率是取平衡用的，和方差起的作用差不多，比如有两个跑步的人，一个人起速是10m/s、其10s后为20m/s；另一个人起速是1m/s、其1s后为2m/s。如果仅仅算差值，那么两个差距就很大了，如果使用速度增加率（加速度，即都为1m/s^2）来衡量，两个人就是一样的加速度。因此，C4.5克服了ID3用信息增益选择属性时偏向选择取值多的属性的不足。

3) CART(Classification and Regression Tree)算法

分类与回归树模型由Breiman等人在1984年提出，是应用广泛的决策树学习方法。CART同样由特征选择、树的生成及剪枝组成，既可以用于分类也可以用于回归。CART是在给定输入随机变量X条件下输出随机变量Y的条件概率分布的学习方法。CART假设决策树是二叉树，内部结点特征的取值为“是”和“否”，左分支是取值为“是”的分支，右分支是取值为“否”的分支。这样的决策树等价于递归地二分每个特征，将输入空间（即特征空间）划分为有限个单元，并在这些单元上确定预测的概率分布，也就是输入给定的条件下输出的条件概率分布。

CART算法由以下两步组成。

(1) 决策树生成：基于训练数据集生成决策树，生成的决策树要尽量大。

(2) 决策树剪枝：用验证数据集对已生成的树进行剪枝并选择最优子树，这时用损失函数最小作为剪枝的标准。

CART算法主要分为两大部分：

(1) 回归数的生成，针对Y是连续变量。

(2) 分类树的生成，针对Y是离散变量。

下面用代码来实现一个决策树分类的案例。这次使用的数据是泰坦尼克号的乘客的生还/遇难数据，具体代码如下：

```
#导入pandas库
import pandas as pd
#读取数据
data = pd.read_csv('train.csv')
#查看数据的前几行
data.head()
```

这段代码的输出结果如下：

| | PassengerId | Survived | Pclass | Name | Sex | Age | SibSp | Parch | Ticket | Fare | Cabin | Embarked |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 3 | Braund, Mr. Owen Harris | male | 22.0 | 1 | 0 | A/5 21171 | 7.2500 | NaN | S |
| 1 | 2 | 1 | 1 | Cumings, Mrs. John Bradley (Florence Briggs Th... | female | 38.0 | 1 | 0 | PC 17599 | 71.2833 | C85 | C |
| 2 | 3 | 1 | 3 | Heikkinen, Miss. Laina | female | 26.0 | 0 | 0 | STON/O2. 3101282 | 7.9250 | NaN | S |
| 3 | 4 | 1 | 1 | Futrelle, Mrs. Jacques Heath (Lily May Peel) | female | 35.0 | 1 | 0 | 113803 | 53.1000 | C123 | S |
| 4 | 5 | 0 | 3 | Allen, Mr. William Henry | male | 35.0 | 0 | 0 | 373450 | 8.0500 | NaN | S |

从这段代码可以看到该数据共有12个特征，对于每个特征的取值类型也能知道，这便为进行下一步分析打下了基础。

```
#查看数据说明
data.info()
```

了解数据，结果如下：

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
PassengerId     891 non-null int64
Survived        891 non-null int64
Pclass          891 non-null int64
Name            891 non-null object
Sex             891 non-null object
Age             714 non-null float64
SibSp           891 non-null int64
Parch           891 non-null int64
Ticket          891 non-null object
Fare            891 non-null float64
Cabin           204 non-null object
Embarked        889 non-null object
dtypes: float64(2), int64(5), object(5)
    #对'Age'这一列进行缺失值的填充,采用均值填充
    data['Age'].fillna(data['Age'].mean(),inplace=True)
    #根据分析,'Cabin'这一列数据不是想要的特征,所以不对其进行填充
    #处理后再对数据进行描述
    data.describe()
```

对存在缺失值的列进行数值填充，采用均值填充，结果如下：

| | PassengerId | Survived | Pclass | Age | SibSp | Parch | Fare |
|---|---|---|---|---|---|---|---|
| **count** | 891.000000 | 891.000000 | 891.000000 | 891.000000 | 891.000000 | 891.000000 | 891.000000 |
| **mean** | 446.000000 | 0.383838 | 2.308642 | 29.699118 | 0.523008 | 0.381594 | 32.204208 |
| **std** | 257.353842 | 0.486592 | 0.836071 | 13.002015 | 1.102743 | 0.806057 | 49.693429 |
| **min** | 1.000000 | 0.000000 | 1.000000 | 0.420000 | 0.000000 | 0.000000 | 0.000000 |
| **25%** | 223.500000 | 0.000000 | 2.000000 | 22.000000 | 0.000000 | 0.000000 | 7.910400 |
| **50%** | 446.000000 | 0.000000 | 3.000000 | 29.699118 | 0.000000 | 0.000000 | 14.454200 |
| **75%** | 668.500000 | 1.000000 | 3.000000 | 35.000000 | 1.000000 | 0.000000 | 31.000000 |
| **max** | 891.000000 | 1.000000 | 3.000000 | 80.000000 | 8.000000 | 6.000000 | 512.329200 |

```
X = data[['Pclass', 'Sex', 'Age']]
y = data['Survived']
```

构建 label 和特征，进行模型训练：

```
#数据分割
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25, random_state = 33)

#使用 sklearn.feature_extraction 中的特征转换器
from sklearn.feature_extraction import DictVectorizer
vec = DictVectorizer(sparse = False)

#特征转换后，发现凡是类别型的特征都单独剥离出来，独成一项特征，数值型的则保持不变
X_train = vec.fit_transform(X_train.to_dict(orient = 'record'))
#同样对测试数据进行特征转换
X_test = vec.fit_transform(X_test.to_dict(orient = 'record'))

#导入决策树分类器
from sklearn.tree import DecisionTreeClassifier
#初始化分类器
dtc = DecisionTreeClassifier()
#用训练数据进行学习
dtc.fit(X_train, y_train)

#对测试集数据进行预测
y_pred = dtc.predict(X_test)

#对算法进行评价
from sklearn.metrics import classification_report
#输出预测准确性
print(dtc.score(X_test, y_test))
#输出更加详细的分类性能
print (classification_report(y_pred, y_test, target_names = ['die', 'survived']))
```

输出结果如下：

```
0.834080717489
              precision    recall  f1-score   support

        die       0.90      0.84      0.87       143
   survived       0.74      0.82      0.78        80

avg / total       0.84      0.83      0.84       223
```

该模型在该数据上的总体预测准确率为83.4%，但是对遇难者的预测准确率达到了90%，对幸存者的预测准确率只有74%，这说明该模型还存在提高的空间。

特点分析：与其他的模型相比，决策树算法在模型描述上有着巨大的优势。决策树的推断逻辑非常直观，具有清晰的可解释性，也方便了模型的可视化。这些特性也保证了使用决策树模型时是无须考虑对数据的量化或标准化的，并且决策树仍属于有参数的学习模型，需要花费很多时间在训练数据上的。

但是决策树很容易产生过拟合的问题，过拟合指的是在训练集上表现良好，而在测试集上表现很差的现象。产生过拟合主要有以下几个原因。

(1) 噪音数据：训练数据中存在噪音数据，决策树的某些结点由噪音数据作为分割标准，导致决策树无法代表真实数据。

(2) 缺少代表性数据：训练数据没有包含所有具有代表性的数据，导致某一类数据无法很好的匹配，这一点可以通过观察混淆矩阵(Confusion Matrix)分析得出。

(3) 多重比较(Mulitple,Comparition)：这一情况和决策树选取分割点类似，需要在每个变量的每个值中选取一个作为分割的代表，所以选出一个噪音分割标准的概率是很大的。

决策树防止过拟合的一个有效操作是修枝剪叶。

决策树过拟合往往是因为太过"茂盛"，也就是结点过多，所以需要裁剪(Prune Tree)枝叶。裁剪枝叶的策略对决策树正确率的影响很大，主要有以下两种裁剪策略。

(1) 前置裁剪：在构建决策树的过程中提前停止，那么会将切分结点的条件设置得很苛刻，导致决策树很短小，结果就是决策树无法达到最优。实践证明这种策略无法得到较好的结果。

（2）后置裁剪：决策树的剪枝往往通过极小化决策树整体的损失函数或代价函数来实现。这样考虑了减小模型复杂度，决策树生成学习局部模型，而决策树剪枝学习整体的模型，利用损失函数最小原则进行剪枝就是用正则化的极大似然估计进行模型选择。

设一组叶结点回缩到其父结点之前的整体树的损失函数值比之后的函数值要大，则进行剪枝。这个过程一直进行，直到不能继续为止，最后得到损失函数最小的子树。

## 本章小结

（1）本章介绍了机器学习的一般流程。

（2）介绍有监督学习的几个常用算法，有监督学习包括线性回归、Logistic 回归、朴素贝叶斯、SVM、KNN 和决策树等。

（3）本章从原理到应用介绍了几个常用的机器学习算法，并分析了它们各自的优缺点。

（4）作为全书最核心的章节之一，本章较为详细地从 Python 代码的角度教大家怎样使用这些算法。

## 习题

1. 对泰坦尼克号的船员生存/遇难数据用 Logistic 回归做分类，比较它与决策树算法的准确率。

2. 尝试对泰坦尼克号的船员生存/遇难数据选用不同的特征做预测，比较它们的准确率。

3. 简述朴素贝叶斯算法的原理。

4. 比较使用不同的核函数时 SVM 算法的效果。

5. 尝试用不同的决策树算法对书中的数据进行分类，比较它们的准确率。

# 第 11 章

# 机器学习——无监督学习

**本章学习目标：**

- 了解无监督机器学习原理
- 了解聚类问题相关算法并进行运用
- 了解关联规则问题相关算法并进行运用

## 11.1 无监督学习

第 10 章重点介绍了监督学习。在监督学习中，必须具有一定的先验知识，例如人工标注类别。但是在现实生活中，并非所有数据都会具有先验知识，往往缺乏这类人工标注，或者人工标注成本过高。这时就需要在没有人工参与的情况下让计算机自主地基于某种算法对数据进行处理和学习，这称为无监督学习。在无监督学习中，数据并不被特别标识，学习模型是为了推断出数据的一些内在结构，往往在结果出来之前没人知道结果是什么样子。其常见的应用场景有聚类和关联规则的学习等。聚类算法的典型应用包括鸡尾酒会问题(即在一个鸡尾酒会上有两种声音，被两个不同的麦克风在不同的地方接收到，如何分离这两种不同的声音?)、图片的自动分类和识

别、社交媒体用户的人群划分等。图 11.1 展示了本章将涉及的几种无监督机器学习算法。

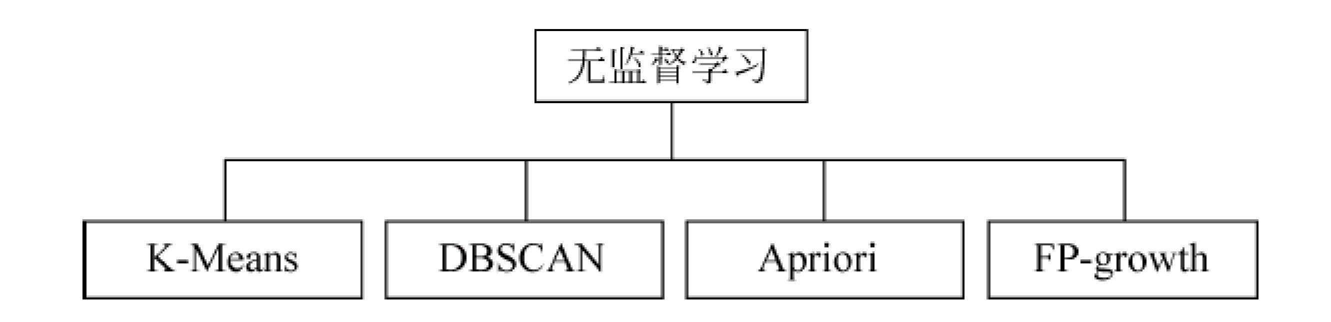



图 11.1　本章涉及的机器学习算法

## 11.2　聚类

在正式讨论聚类之前，需要先弄清楚一个问题：如何定量计算两个可比较元素间的相异度。用通俗的话说，相异度就是两个东西差别有多大，例如人类与章鱼的相异度明显大于人类与黑猩猩的相异度，这是我们能直观感受到的。但是，计算机没有这种直观感受能力，因此必须对相异度在数学上进行定量定义。

设 $X=(x_1,x_2,\cdots,x_n)$，$Y=(y_1,y_2,\cdots,y_n)$，其中 X、Y 是两个元素项，各自具有 n 个可度量特征属性，那么 X 和 Y 的相异度定义为 $d(X,Y)=f(X,Y)\rightarrow R$，其中 R 为实数域。也就是说，相异度是两个元素对实数域的一个映射，所映射的实数定量表示两个元素的相异度。下面介绍不同类型变量相异度的计算方法。

### 11.2.1　相异度

#### 1. 标量

标量也就是无方向意义的数字，也叫标度变量。现在先考虑元素的所有特征属性都是标量的情况。例如，计算 $X=\{2,1,102\}$ 和 $Y=\{1,3,2\}$ 的相异度。一种很自然的想法是用两者的欧几里得距离来作为相异度，欧几里得距离的定义如下：

$$d(X,Y)=\sqrt{(x_1-y_1)^2+(x_2-y_2)^2+\cdots+(x_n-y_n)^2} \tag{11.1}$$

其意义就是两个元素在欧氏空间中的集合距离，因为其直观易懂且可解释性强，被广泛用于标识两个标量元素的相异度。将上面两个示例数据代入公式，可得两者的欧氏距离为：

$$d(X,Y)=\sqrt{(2-1)^2+(1-3)^2+(102-2)^2}=100.025 \tag{11.2}$$

除欧氏距离外,常用作度量标量相异度的还有曼哈顿距离和闵可夫斯基距离,两者的定义如下。

曼哈顿距离:

$$d(X,Y)=|x_1-y_1|+|x_2-y_2|+\cdots+|x_n-y_n| \tag{11.3}$$

闵可夫斯基距离:

$$d(X,Y)=\sqrt[p]{|x_1-y_1|^p+|x_2-y_2|^p+\cdots+|x_n-y_n|^p} \tag{11.4}$$

欧氏距离和曼哈顿距离可以看作是闵可夫斯基距离在 $p=2$ 和 $p=1$ 下的特例。另外,这 3 种距离都可以加权,这很容易理解,因此不再赘述。

下面要说一下标量的规格化问题。上面这样计算相异度的方式有一点问题,就是取值范围大的属性对距离的影响高于取值范围小的属性。例如上述例子中第 3 个属性的取值跨度远大于前两个,这样不利于真实反映相异度,为了解决这个问题,一般要对属性值进行规格化。所谓规格化,就是将各个属性值按比例映射到相同的取值区间,这样是为了平衡各个属性对距离的影响。通常将各个属性映射到[0,1]区间,映射公式为:

$$a_i'=\frac{a_i-\min(a_i)}{\max(a_i)-\min(a_i)} \tag{11.5}$$

其中,$\max(a_i)$和 $\min(a_i)$表示所有元素项中第 i 个属性的最大值和最小值。例如将示例中的元素规格化到[0,1]区间后,就变成了 $X'=\{1,0,1\}$,$Y'=\{0,1,0\}$,重新计算欧氏距离约为 1.732。

### 2. 二元变量

二元变量是只能取 0 和 1 两种值的变量,有点类似布尔值,通常用来标识是或不是这种二值属性。对于二元变量,上面提到的距离不能很好地标识其相异度,因此需要一种更适合的标识。一种常用的方法是用元素相同序位同值属性的比例来标识其相异度。

设有 $X=\{1,0,0,0,1,0,1,1\}$,$Y=\{0,0,0,1,1,1,1,1\}$,可以看到两个元素的第 2、3、5、7、8 个属性取值相同,而第 1、4、6 个取值不同,那么相异度可以标识为 3/8=0.375。一般地,对于二元变量,相异度可用"取值不同的同位属性数/单个元素的属性位数"标识。

上面所说的相异度应该叫对称二元相异度。在现实中还有一种情况,就是我们

只关心两者都取1的情况，而认为两者都取0的属性并不意味着两者更相似。例如在根据病情对病人聚类时，如果两个人都患有肺癌，则认为两个人增强了相似度，但如果两个人都没患肺癌，并不觉得这加强了两人的相似性，在这种情况下，改用"取值不同的同位属性数/(单个元素的属性位数-同取0的位数)"来标识相异度，这叫非对称二元相异度。如果用1减去非对称二元相异度，则得到非对称二元相似度，也叫Jaccard系数，这是一个非常重要的概念。

**3. 分类变量**

分类变量是二元变量的推广，类似于程序中的枚举变量，但各个值没有数字或序数意义，例如颜色、民族等。对于分类变量，用"取值不同的同位属性数/单个元素的全部属性数"来标识其相异度。

**4. 序数变量**

序数变量是具有序数意义的分类变量，通常可以按照一定的顺序意义排列，例如冠军、亚军和季军。对于序数变量，一般为每个值分配一个数，叫这个值的秩，然后以秩代替原值当作标量属性计算相异度。

**5. 向量**

对于向量，由于它不仅有大小而且有方向，所以闵可夫斯基距离不是度量其相异度的好办法，一种流行的做法是用两个向量的余弦度量，其度量公式为：

$$s(X,Y)=\frac{X^{t}Y}{\|X\|\,\|Y\|} \tag{11.6}$$

其中，$\|X\|$表示X的欧几里得范数。注意，余弦度量度量地不是两者的相异度，而是相似度。

讨论完了相异度计算的问题，就可以正式定义聚类问题了。

所谓聚类问题，就是给定一个元素集合D，其中每个元素具有n个可观察属性，使用某种算法将D划分成k个子集，要求每个子集内部的元素之间相异度尽可能低，而不同子集的元素相异度尽可能高。其中每个子集叫一个簇。

与分类不同，分类是示例式学习，要求在分类前明确各个类别，并断言每个元素映射到一个类别，而聚类是观察式学习，在聚类前可以不知道类别甚至不给定类别数

量，是无监督学习的一种。目前，聚类广泛应用于统计学、生物学、数据库技术和市场营销等领域，相应的算法也非常多。下节介绍一种最简单的聚类算法——K 均值(K-Means)算法。

通常，人们根据样本间的某种距离或者相似性来定义聚类，即把相似的(或距离近的)样本聚为同一类，而把不相似的(或距离远的)样本归在其他类。

## 11.2.2 K-Means 算法

视频讲解

### 1. 算法简介

K-Means 算法是一种聚类算法。所谓聚类，即根据相似性原则，将具有较高相似度的数据对象划分至同一类簇，将具有较高相异度的数据对象划分至不同类簇。聚类与分类最大的区别在于，聚类过程为无监督过程，即待处理数据对象没有任何先验知识，而分类过程为有监督过程，即存在有先验知识的训练数据集。

K-Means 算法中的 K 代表类簇个数，means 代表类簇内数据对象的均值(这种均值是一种对类簇中心的描述)。K-Means 算法是一种基于划分的聚类算法，以距离作为数据对象间相似性度量的标准，即数据对象间的距离越小，它们的相似性越高，则它们越有可能在同一个类簇。数据对象间距离的计算有很多种，K-Means 算法通常采用欧氏距离来计算数据对象间的距离。

K-Means 算法是一种很常见的聚类算法，它的基本思想是通过迭代寻找 K 个聚类的一种划分方案，使得用这 K 个聚类的均值来代表相应各类样本时所得的总体误差最小。

K-Means 算法的基础是最小误差平方和准则。其代价函数是：

$$J(c,\mu)=\sum_{i=1}^{K}\|x(i)-\mu_c(i)\|^2 \tag{11.7}$$

式中，$\mu_c(i)$表示第 i 个聚类的均值。通常希望代价函数最小，直观地说，各类内的样本越相似，其与该类均值间的误差平方越小，对所有类所得到的误差平方求和，即可验证分为 K 类时各聚类是否为最优的。

上式的代价函数无法用解析的方法最小化，只能有迭代的方法。K-Means 算法是将样本聚类成 K 个簇(cluster)，其中 K 是用户给定的，然后通过迭代的方法达到误差最小或达到可接受的误差范围内，其求解过程非常直观、简单，具体算法描述

如下：

首先初始化 K 个类簇中心；然后计算各个数据对象到聚类中心的距离，把数据对象划分至距离其最近的聚类中心所在的类簇中；接着根据所得类簇更新类簇中心；然后继续计算各个数据对象到聚类中心的距离，把数据对象划分至距离其最近的聚类中心所在的类簇中；接着根据所得类簇继续更新类簇中心；一直迭代，直到达到最大迭代次数 TT，或者两次迭代 JJ 的差值小于某一阈值时迭代终止，得到最终聚类结果。

### 2. K-Means 算法的 Python 实现

(1) 导入需要用到的库。

```
#导入需要用到的库
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
```

(2) 定义一个读取文件的函数，方便后面读取文件。

```
#定义一个读取文件的函数
def load_data(input_file):
    X = []
    with open(input_file, 'r') as f:
        for line in f.readlines():
            data = [float(x) for x in line.split(',')]
            X.append(data)

    return np.array(X)
```

(3) 读取文件，并定义集群的数量。

```
#读取文件
data = load_data('F:\data\dataset\Chapter04\data_multivar.txt')
num_clusters = 4
```

(4) 将文件中的数据可视化展示，能让用户对数据有一个直观的认识。

```
#将数据可视化表示
plt.figure()
```

```
plt.scatter(data[:,0], data[:,1], marker = 'o',
         facecolors = 'none', edgecolors = 'k', s = 30)
x_min, x_max = min(data[:, 0]) - 1, max(data[:, 0]) + 1
y_min, y_max = min(data[:, 1]) - 1, max(data[:, 1]) + 1
plt.title('Input data')
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.xticks(())
plt.yticks(())
plt.show()
```

运行上述代码,可以看到如图 11.2 所示的聚类数据分布图。

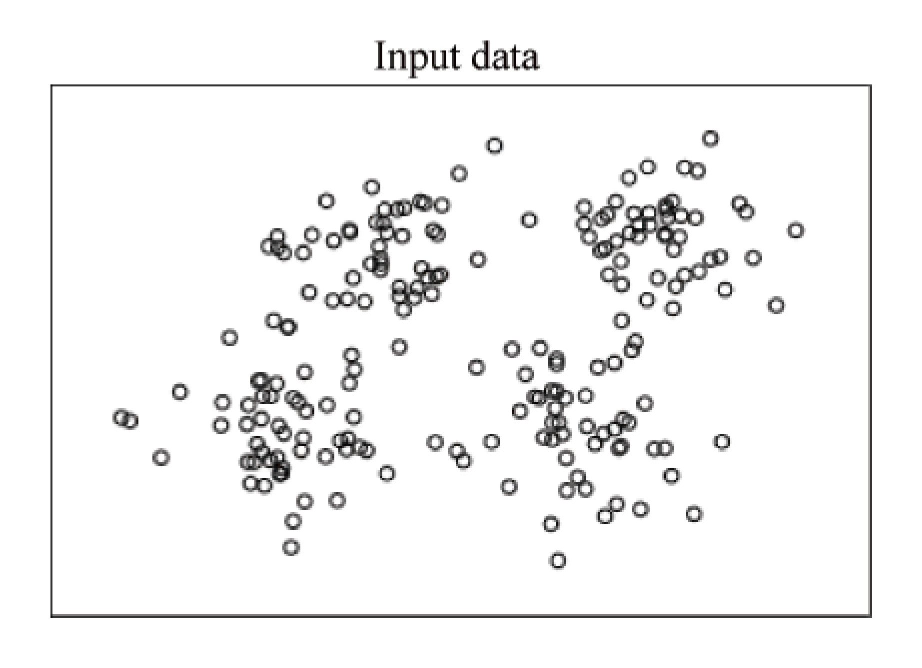



图 11.2　聚类数据分布图

(5) 下面训练模型,先初始化一个 K-Means 算法模型,然后训练。

```
kmeans = KMeans(init = 'k - means++', n_clusters = num_clusters, n_init = 10)
kmeans.fit(data)
```

(6) 训练完之后对边界进行可视化处理。

```
#设置网格数据的步长
step_size = 0.01

#画出边界
x_min, x_max = min(data[:, 0]) - 1, max(data[:, 0]) + 1
y_min, y_max = min(data[:, 1]) - 1, max(data[:, 1]) + 1
x_values, y_values = np.meshgrid(np.arange(x_min, x_max, step_size), np.arange(y_min, y_
         max, step_size))

#预测网格中所有数据点的标记
predicted_labels = kmeans.predict(np.c_[x_values.ravel(), y_values.ravel()])
```

(7) 模型训练完成之后将结果展示出来,并将聚类中心点画出。

```
#画出结果
predicted_labels = predicted_labels.reshape(x_values.shape)
plt.figure()
plt.clf()
plt.imshow(predicted_labels, interpolation = 'nearest',
           extent = (x_values.min(), x_values.max(), y_values.min(), y_values.max()),
           cmap = plt.cm.Paired,
           aspect = 'auto', origin = 'lower')

plt.scatter(data[:,0], data[:,1], marker = 'o',
        facecolors = 'none', edgecolors = 'k', s = 30)
centroids = kmeans.cluster_centers_
plt.scatter(centroids[:,0], centroids[:,1], marker = 'o', s = 200, linewidths = 3,
        color = 'k', zorder = 10, facecolors = 'black')
x_min, x_max = min(data[:, 0]) - 1, max(data[:, 0]) + 1
y_min, y_max = min(data[:, 1]) - 1, max(data[:, 1]) + 1
plt.title('Centoids and boundaries obtained using KMeans')
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.xticks(())
plt.yticks(())
plt.show()
```

运行代码,展示结果如图 11.3 所示。

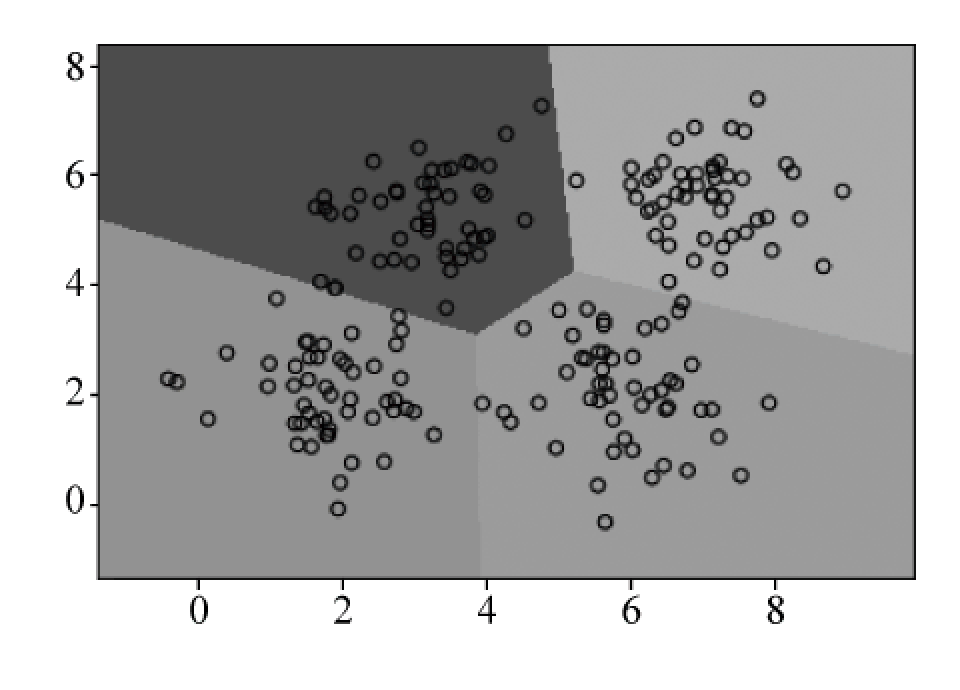



图 11.3 K-Means 输出结果

### 3. K-Means 算法的评价

算法总结:K-Means 算法虽然比较简单,但也有几个比较大的缺点。

(1) K 值的选择是用户指定的,不同的 K 得到的结果会有很大的不同,导致算法效果存在一定的随机性。因此在应用算法前可考虑使用可视化手段大致判断聚类簇的数量,或多尝试不同的 K 值,比较并确定一个更为合理的结果。

(2) K-Means 算法对于球形簇有较好的聚类效果，而对于非球形簇的效果较差。例如对于图 11.4(a)中的数据对象来说，其分布是典型的条状线性分布，合理的聚类结果应如图 11.4(b)所示，分为黑、白两簇，而若使用 K-Means 算法，则聚类将极可能是如图 11.4(c)所示的结果，与我们所设想的结果大相径庭。因此，若已知待聚类对象是非球形簇分布，应考虑使用其他聚类方法，如 11.2.3 节中将介绍的基于密度的算法。

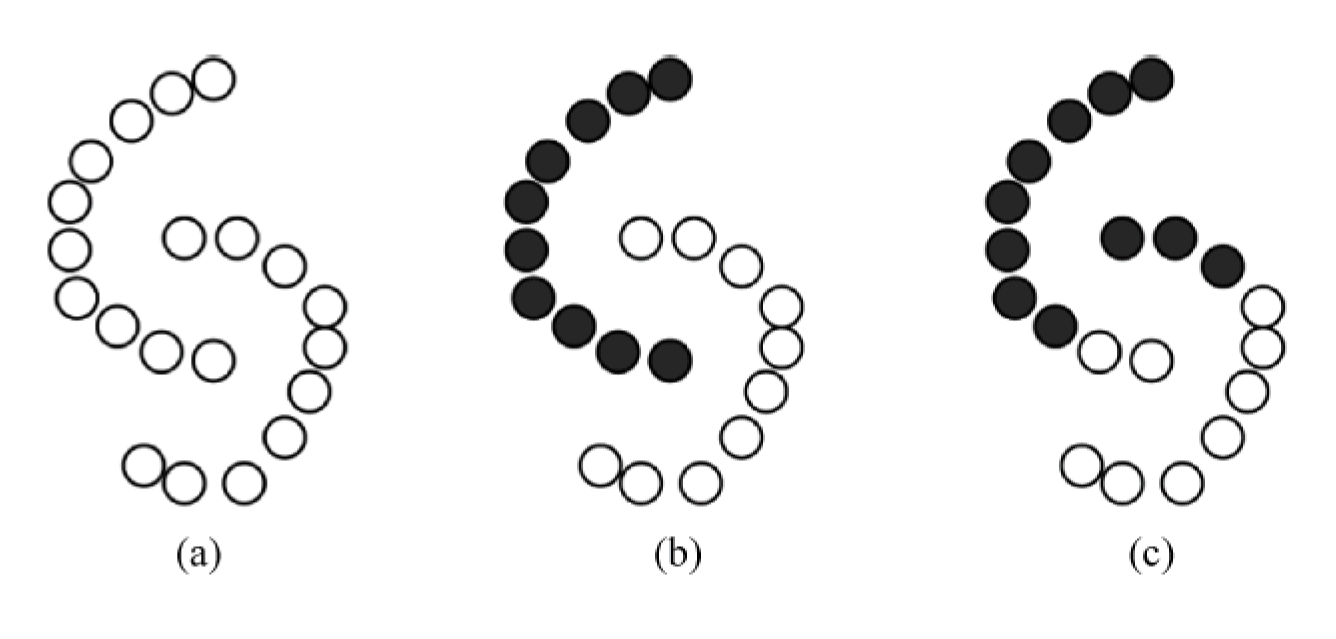



图 11.4　非球形簇示例

(3) K-Means 算法对 K 个初始质心的选择比较敏感，容易陷入局部最小值。例如，如果初始聚类中心较为接近，选在同一个聚类簇上，结果必然与理论上的合理值相异。有一些改进的 K-Means 算法在尝试解决这样的问题，这类算法通常对初始中心点的选取比较严格，各中心点的距离较远，这就避免了初始聚类中心会选到一个类上，在一定程度上克服了算法陷入局部最优状态。

二分 K-Means(Bisecting K-Means)算法是其中一种，其主要思想是首先将所有点作为一个簇，然后将该簇一分为二。之后选择能最大限度降低聚类代价函数(也就是误差平方和)的簇划分为两个簇。以此进行下去，直到簇的数目等于用户给定的数目 K 为止。以上隐含的一个原则是：因为聚类的误差平方和能够衡量聚类性能，该值越小表示数据点越接近于它们的质心，聚类效果就越好。所以需要对误差平方和最大的簇进行再一次划分，因为误差平方和越大，表示该簇聚类效果越不好，越有可能是多个簇被当成了一个簇，首先需要对这个簇进行划分。下面是二分 K-Means 聚类的伪代码：

```
初始化:所有点作为一个簇
IF 簇的数目小于 K
    FOR 每一个簇
        计算总误差
        在给定的簇上进行 K-Means 聚类(K=2)
```

计算二分后的总误差
选择使误差最小的簇进行划分

### 11.2.3 DBSCAN算法

视频讲解

#### 1. 算法简介

DBSCAN(Density-Based Spatial Clustering of Applications with Noise,具有噪声的基于密度的聚类方法)是典型的基于密度的聚类算法。与K-Means算法相比,DBSCAN算法在执行之初不需要预先指定聚类簇的个数。当然,最终的聚类簇个数在结果出来之前也就不得而知了。

DBSCAN算法的优点是聚类速度快、能够有效处理噪声点以及聚类簇的形状没有偏倚。其缺点是当数据量增大时,要求较大的内存支持,I/O消耗也很大,同时当空间聚类的密度不均匀、聚类间距差相差很大时聚类质量较差。

既然是基于密度的聚类,这里有必要给出与密度相关的一些定义,以便后续讨论具体算法。

- ε-邻域:给定对象O半径ε内的区域称为该对象的ε-邻域。
- 核心对象:如果给定对象的ε-邻域内的样本点数大于或等于MinPts,则称该对象为核心对象。MinPts为人为预先指定的阈值参数。
- 直接密度可达:给定一个对象集合D,如果p在q的ε-邻域内,且q是一个核心对象,则说对象p从对象q出发是直接密度可达的。显然,对象p是从另一个对象q直接密度可达的,当且仅当q是核心对象,并且p在q的ε-邻域中。
- 密度可达:对于样本集合D,如果存在一个对象链$p_1,p_2,\cdots,p_n$,使得$p_1=q$,$p_n=p$,并且$p_i\in D(1\leqslant i\leqslant n)$,那么$p_{i+1}$是从$p_i$关于ε和MinPts直接密度可达的。
- 密度相连:如果存在对象$q\in D$,使对象$p_1$和$p_2$都是从q关于ε和MinPts密度可达的,那么对象$p_1$、$p_2$是关于ε和MinPts密度相连的。

有了上述概念,就可以对DBSCAN聚类进行定义了:由密度可达关系导出的最大密度相连的样本集合即为最终聚类簇。

这个 DBSCAN 的簇里面可以有一个或者多个核心对象。如果只有一个核心对象，则簇里其他的非核心对象样本都在这个核心对象的 ε-邻域里；如果有多个核心对象，则簇里的任意一个核心对象的 ε-邻域中一定有一个其他的核心对象，否则这两个核心对象无法密度可达。这些核心对象的 ε-邻域里所有样本的集合组成一个 DBSCAN 聚类簇。

那么怎么才能找到这样的簇样本集合呢？DBSCAN 使用的方法很简单，它任意选择一个没有类别的核心对象作为种子，然后找到所有这个核心对象能够密度可达的样本集合，即为一个聚类簇。接着继续选择另一个没有类别的核心对象去寻找密度可达的样本集合，这样就得到另一个聚类簇。一直运行到所有核心对象都有类别为止。

此外，对于 DBSCAN 算法有 3 个问题需要注意。

第一个是一些异常样本点或者说少量游离于簇外的样本点，这些点不在任何一个核心对象的周围，在 DBSCAN 中，一般将这些样本点标记为噪音点。

第二个是距离的度量问题，即如何计算某样本和核心对象样本的距离。在 DBSCAN 中，一般采用最近邻思想，采用某一种距离度量来衡量样本距离，例如欧式距离。

第三个问题比较特殊，某些样本到两个核心对象的距离可能都小于 ε，但是这两个核心对象由于不是密度直达，又不属于同一个聚类簇，那么如何界定这个样本的类别呢？一般来说，此时 DBSCAN 采用先来后到，先进行聚类的类别簇会标记这个样本为它的类别。也就是说，DBSCAN 算法不是完全稳定的算法。

下面给出 DBSCAN 的伪代码：

```
输入:
    D:一个包含 n 个对象的数据集
    ε:半径参数
    MinPts:邻域密度阈值
输出:基于密度的簇的集合
方法:
标记所有对象为 unvisited;
DO
    随机选择一个 unvisited 对象 p;
    标记 p 为 visited
    IF p 的 ε - 邻域至少有 MinPts 个对象
        创建一个新簇 C,并把 p 添加到 C
        令 N 为 p 的 ε - 邻域中的对象的集合;
```

```
            FOR N 中的每个点 p'
                IF p'是 unvisited
                    标记 p'为 visited
                    IF p'的 ε - 邻域至少有 MinPts 个点,把这些点添加到 N
                IF p'还不是任何簇的成员,把 p'添加到 C;
            END FOR
            输出 C;
        ELSE 标记 p 为噪声;
    UNTIL 没有标记为 unvisited 的对象
```

### 2. DBSCAN 算法的 Python 实现

在 Scikit-learn 库中已经集成有 DBSCAN 算法的实现,具体由 cluster 模块中的 DBSCAN 类进行处理。DBSCAN 类的初始化需要最多 8 个参数(均为可选参数),其中几个比较重要的参数如下。

(1) eps:同一个簇中样本的最大距离,默认为 0.5。一般需要在多组值里面选择一个合适的阈值。如果 eps 过大,更多的点会落在核心对象的 ε-邻域,此时类别数可能会减少,本来不应该是一类的样本也会被划为一类。反之,类别数可能会增大,本来是一类的样本却被划分开。

(2) min_samples:一个簇中至少需要包含的样本数,默认为 5。一般需要在多组值里面选择一个合适的阈值。它通常和 eps 一起调参。在 eps 一定的情况下,如果 min_samples 过大,则核心对象会过少,此时簇内部分本来是一类的样本可能会被标为噪音点,类别数也会变多。反之,如果 min_samples 过小,则会产生大量的核心对象,可能会导致类别数过少。

(3) metric:距离公式,用户可以用默认的欧式距离,还可以自己定义距离函数。

(4) algorithm:最近邻搜索算法参数,可选值包括 auto、ball_tree、kd_tree、brute,默认为 auto。在可选值中,brute 是蛮力实现,kd_tree 是 KD 树实现,ball_tree 是球树实现,auto 则会在 3 种算法中做权衡,选择一个拟合最好的最优算法。在一般情况下使用默认的 'auto' 就够了。如果数据量很大或者特征很多,用 'auto' 建树时间可能会很长,效率不高,建议选择 KD 树实现 'kd_tree',此时如果发现 'kd_tree' 速度比较慢或者已经知道样本分布不是很均匀,可以尝试用 'ball_tree'。如果输入样本是稀疏的,无论选择哪个算法,最后实际运行的都是 'brute'。

关于 DBSCAN 的使用代码如下:

```
import numpy as np

from sklearn.cluster import DBSCAN
from sklearn import metrics
from sklearn.datasets.samples_generator import make_blobs
from sklearn.preprocessing import StandardScaler

#生成示例数据
centers = [[1, 1], [-1, -1], [1, -1]]
X, labels_true = make_blobs(n_samples = 750, centers = centers, cluster_std = 0.4,
                            random_state = 0)
X = StandardScaler().fit_transform(X)

#DBSCAN 聚类
db = DBSCAN(eps = 0.3, min_samples = 10).fit(X)
core_samples_mask = np.zeros_like(db.labels_, dtype = bool)
core_samples_mask[db.core_sample_indices_] = True
labels = db.labels_

n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
n_noise_ = list(labels).count(-1)

print('Estimated number of clusters: %d' % n_clusters_)
print('Estimated number of noise points: %d' % n_noise_)

#结果可视化
import matplotlib.pyplot as plt

unique_labels = set(labels)
colors = [plt.cm.Spectral(each)
          for each in np.linspace(0, 1, len(unique_labels))]
for k, col in zip(unique_labels, colors):
    if k == -1:
        #Black used for noise.
        col = [0, 0, 0, 1]

    class_member_mask = (labels == k)

    xy = X[class_member_mask & core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor = tuple(col),
             markeredgecolor = 'k', markersize = 14)

    xy = X[class_member_mask & ~core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor = tuple(col),
             markeredgecolor = 'k', markersize = 6)

plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()
```

代码运行的结果如图 11.5 所示。

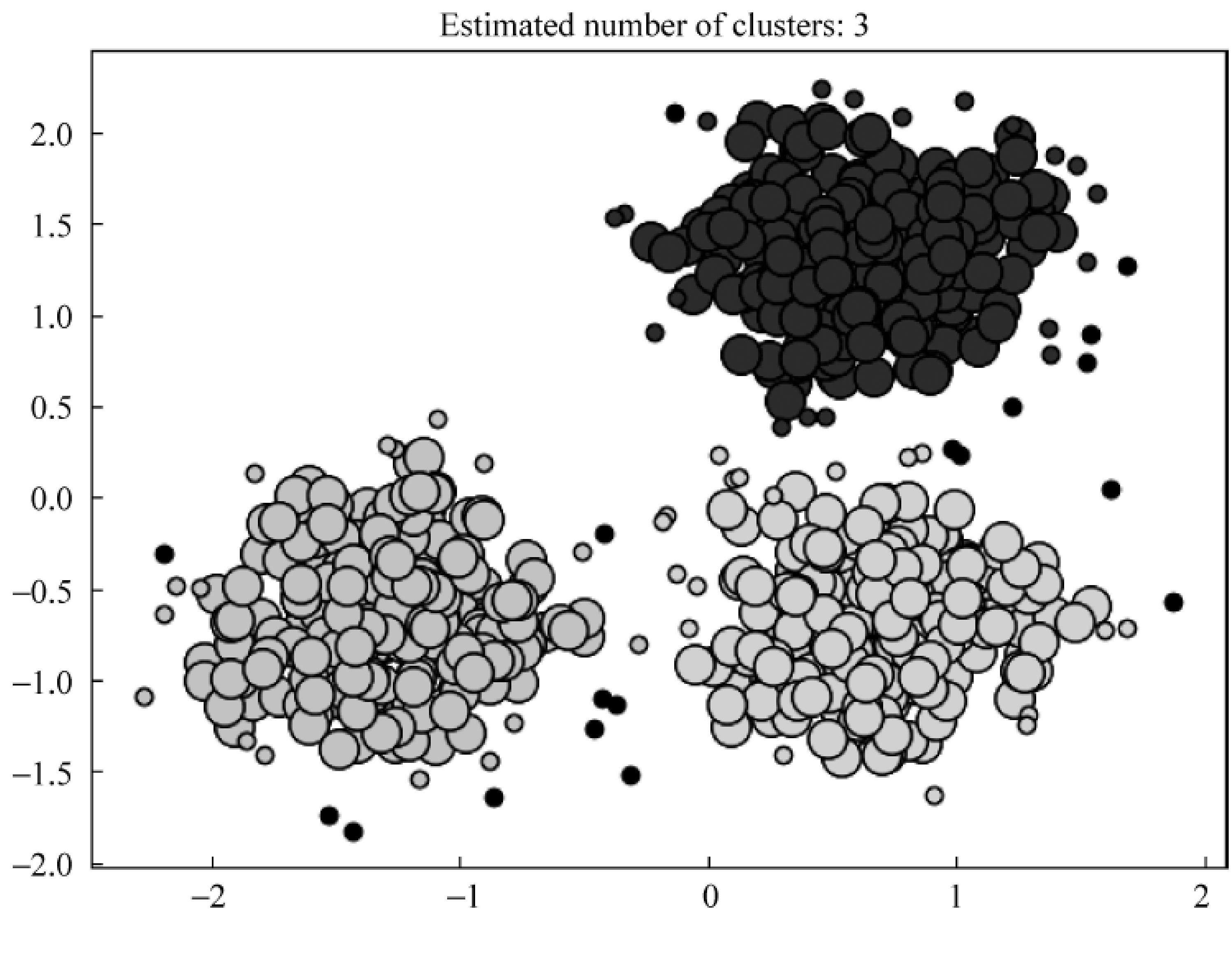



图 11.5　DBSCAN 聚类结果

## 11.3　关联规则

### 11.3.1　关联分析

关联分析最早来源于购物篮分析。对超市的管理者来说，一个重要的信息是顾客在一次购物中会同时购买何种商品，一旦知悉该信息，那么在产品促销、商品摆放、商品进货等决策环节便可做到有的放矢。一个典型的案例便是“尿布与啤酒”的故事：美国一家超市在对其原始交易数据进行数据挖掘时意外地发现，在购买记录中尿布与啤酒这两种截然不同的商品竟然经常被一起购买！后来经过调查和分析才发现这其中蕴含着一种美国人的行为模式，即先生们在下班时会被太太要求去超市为小孩购买尿布，而先生们在购买尿布之后又会顺手购买自己喜爱的啤酒。根据这个信息，超市便大可将尿布与啤酒的货架摆放得尽可能近，又可在促销策略中将两种商品进行捆绑销售。

关联分析的主要目的即是从大量数据中发现事物之间有趣的关联和相关联系。以表 11.1 为例，在这里把顾客的每一次交易称为一个“事务”，记为 T；把所有事务的总和称为“事务集”，记为 D，表 11.1 中的事务集共有 5 个事务；把每一种商品称为一

个“项”，记为 I；把包含多个项的集合称为“项集”，在一个项集里可以有一个项、多个项，甚至零个项。一般的，若一个项集中有 k 个项，则称此项集为 k 项集，例如项集{鸡蛋}是 1 项集，项集{鸡蛋，番茄}是 2 项集，表 11.1 所示的例子中共有 1 个 2 项集，4 个 4 项集。某个项集在整个事务集中出现的次数被称为项集的“出现频度”，在某些文献中也简称为频度或计数。若某个项集在事务集中频繁出现，该项集就值得引起注意。例如在表 11.1 中，项集{鸡蛋}的频度为 4，而项集{鸡蛋，番茄}的频度为 3。

**表 11.1　某菜市场交易清单示例**

| 编　号 | 商　品 | 编　号 | 商　品 |
|---|---|---|---|
| 1 | 鸡蛋，番茄，白菜，萝卜 | 4 | 鸡蛋，猪肉，鱼，番茄 |
| 2 | 猪肉，萝卜，鱼，葱 | 5 | 鸡蛋，葱 |
| 3 | 鸡蛋，猪肉，番茄，白菜 | | |

关联分析中的一个主要概念是关联规则，这是一个形如 X→Y 的蕴含式，X 和 Y 分别为一个项集，其中 X 称为关联规则的先导，Y 称为关联规则的后继。直观地理解就是：当 X 被购买时，Y 也“可能”同时被购买。例如关联规则“尿布→啤酒”表示购买尿布的人也同时会购买啤酒。值得注意的是关联规则有先后顺序之分，购买尿布的人也会顺便购买啤酒，但是购买啤酒的人并不一定会购买尿布。另外，关联规则的后继是“可能”发生，这里的可能性反映了该关联规则的强弱，通常我们只对那些“很可能”发生（或者说频繁发生）的关联规则感兴趣。有两个常用指标用来指示一个关联规则的强弱，即支持度（Support）和置信度（Confidence）。

支持度表示关联规则 X→Y 中 X 和 Y 在事务集 D 中同时出现的概率，即：

$$\mathrm{support}(X \rightarrow Y) = P(X \cap Y)$$

例如某超市总共有 100 人购买了商品，其中 20 人同时购买了牛肉和牛奶，则牛肉→牛奶和牛奶→牛肉的支持度均为 0.2。在表 11.1 所示的例子中，鸡蛋→番茄的支持度为 3/5。由其定义可知，对于支持度来说，其规则的先导和后继没有顺序要求，X→Y 和 Y→X 的支持度总是相等，因此支持度也可针对项集而言，即项集的支持度定义为项集中各项同时出现的概率。若某项集的支持度足够大，例如大于某个我们预先定义的阈值（至于该阈值到底定为多少较为合理，取决于数据和应用的实际情况，应由分析人员基于此根据经验斟酌而定），则将该项集称为“频繁项集”。

置信度是针对一条关联规则来定义的，关联规则 X→Y 的置信度是事务集 D 中包含 X 的同时也包含 Y 的概率，即 X 被购买的条件下 Y 也被购买的概率，如下：

$$\text{confidence}(X \rightarrow Y) = P(Y \mid X)$$

在表 11.1 所示的示例中，番茄→鸡蛋的置信度为 1，即所有买番茄的人最终都同时买了鸡蛋，而鸡蛋→番茄的置信度为 3/4，即在 4 个买鸡蛋的人中有 3 人买了番茄。因此，如果已知某人买了番茄，则可以推想该人有极大可能（接近 100%）想要购买鸡蛋，若已知某人已买鸡蛋，则其有较大可能（接近 75%）想要购买番茄。

如果人为设定一个最小支持度和最小置信度阈值，一旦某条关联规则的支持度和置信度同时达到阈值，则称其为强规则。例如将最小支持度阈值设为 0.5，将最小置信度阈值设为 0.8，则番茄→鸡蛋显然是一条强关联规则，而强关联规则很大可能是“有趣的”关联关系。

关联规则的挖掘实际上就是计算事务集中各个规则的支持度和置信度，找到两者足够大的规则，即提取强关联规则。而由条件概率公式，有：

$$\text{confidence}(X \rightarrow Y) = P(Y \mid X) = \frac{P(X \cap Y)}{P(X)}$$

$$= \frac{\text{support}(X \cap Y)}{\text{support}(X)} = \frac{\text{count}(X \cap Y)}{\text{count}(X)}$$

可知，若要计算关联规则 X→Y 的置信度，归根结底是计算项集{X，Y}和项集{X}的支持度，而支持度的计算归根结底是计算项集的频度。因此，关联规则的挖掘最终归结于发现频繁项集。那么如何发现频繁项集呢？最简答的方法莫过于针对一个事务集中的所有项，基于排列组合生成所有可能的项集，对每一个项集统计其出现次数，从而掌握其频繁程度。但是该方法的缺陷也十分明显，就是当事务集的项数太多时，频度计算的计算量将极其庞大。例如，若商店有 100 种商品（事实上一般商店的商品数远超过该数字），则这些商品可能的项集组合约有 $1.26 \times 10^{30}$ 种之多，即使是现代的计算机，对于这样的计算量仍然需要很长时间才能够完成。

事实上，所有关联规则挖掘算法的核心目的都是如何降低计算量，以便在合理的可接受的时间内得到结果。接下来介绍两种常用的关联规则挖掘算法——Apriori 算法和 FP-growth 算法。

### 11.3.2 Apriori 算法


视频讲解

#### 1. 算法简介

Apriori 这个词本身是“先验”的意思。在定义和解决问题时，有时会

基于一些已知的知识或假设，以便简化问题。Apriori 算法就使用了一条先验知识，从而在计算频繁项集时大大缩减了计算量。那么什么是先验知识呢？该条先验知识的表述是：若某个项集是频繁的，那么它的所有子集也是频繁的。例如若项集{鸡蛋，番茄}是频繁项集，即同时购买鸡蛋和番茄的人很多，那么购买鸡蛋或番茄的人数必然更多(因为这些人中既包含了同时购买鸡蛋、番茄的人，也包括了只购买鸡蛋或番茄的人)，因此{鸡蛋}和{番茄}必然也是频繁项集。这条先验知识被称为 Apriori 原理。

该原理直观上似乎对寻找频繁项集无用，但是其反过来的说法就大有用处了：若一个项集是非频繁项集，则其所有超集必是非频繁项集。例如若{鸡蛋}是非频繁项集，即购买鸡蛋的人本就不多，那么所有包括了鸡蛋这一商品的项集(例如同时购买鸡蛋和番茄)必然更少，因此不管是{鸡蛋，番茄}还是{鸡蛋，番茄，猪肉}等都是非频繁的了。在计算各项集频度时，若发现鸡蛋是非频繁的，那么所有包括了鸡蛋的项集都不用再去计算了。使用该原理就可以避免项集数目的指数级增长，从而在合理的时间内计算出频繁项集。

关联规则的分析大致分为两步，即发现频繁项集，计算置信度从而发现强关联规则，其中发现频繁项集是关键步骤。Apriori 是发现频繁项集的一种算法，该算法需要两个参数——最小支持度阈值和输入数据集。该算法首先会生成所有 1 项集的列表，接着扫描整个事务集来判断哪些项集满足最小支持度阈值，将不满足的项集去除。然后对剩下来的 1 项集进行排列组合生成 2 项集，再重新扫描事务集，去掉不满足最小支持度的 2 项集。该过程不断重复，直到不再有可能的频繁项集可以构建。该过程的伪代码如下：

```
FOR 数据集中每条事务 T
FOR 每个候选项集 I:
    IF I 是否 T 的子集
        增加 I 的计数值
FOR 每个候选项集:
    IF 其支持度不低于最小支持度阈值
        保留该项集
返回所有频繁项集列表
```

### 2. Apriori 算法的 Python 实现

由于 Scikit-learn 中并没有集成 Apriori 算法，这里需要使用 Python 从底层实

现。首先定义一些辅助函数：

```
def loadDataSet():
    return [[1, 3, 4], [2, 3, 5], [1, 2, 3, 5], [2, 5]]

def createC1(dataSet):
    C1 = []
    for transaction in dataSet:
        for item in transaction:
            if not [item] in C1:
                C1.append([item])
    C1.sort()
    return list(map(frozenset, C1))

def scanD(dataset,Ck,minSupport):
    D = map(set, dataset)
    ssCnt = {}
    for tid in D:
        for can in Ck:
            if can.issubset(tid):
                if not can in ssCnt:
                    ssCnt[can] = 1
                else: ssCnt[can] += 1
    numItems = float(len(dataset))
    retList = []
    supportData = {}
    for key in ssCnt:
        support = ssCnt[key]/numItems
        if support >= minSupport:
            retList.insert(0,key)
        supportData[key] = support
    return retList, supportData
```

上述代码创建了3个辅助函数，其中loadDataSet()创建一个简单的数据集用于测试算法结果。createC1()用于从原始事务集中创建1项集。方法流程是首先构建一个空集合C1，接下来遍历事务集中的所有事务，记录其中的每一个项，如果某个项没有在C1中出现，则以该项构建一个列表，添加到C1中，之所以每个项单独构建一个列表，是为了将来能够做集合操作。scanD()用于扫描事务集，并从候选项集中挑选出满足最小支持度阈值的项集作为频繁项集将其返回，同时返回的还有最频繁项集的支持度，该值在后面步骤中会使用到。

下面测试一下代码：

```
>>> dataSet = loadDataSet()
>>> dataSet
[[1, 3, 4], [2, 3, 5], [1, 2, 3, 5], [2, 5]]

>>> C1 = createC1(dataSet)
>>> C1
[frozenset({1}), frozenset({2}), frozenset({3}), frozenset({4}), frozenset({5})]

>>> L1, suppData0 = scanD(dataSet, C1, 0.5)
>>> L1
[frozenset({5}), frozenset({2}), frozenset({3}), frozenset({1})]
```

可见，对于1项集来说，若将最小支持度设为0.5，则{1}、{2}、{3}、{5} 4项的出现概率均超过半数，因此均为频繁1项集；而{4}仅在两个事务中出现，未超过半数，为非频繁项集。

接下来考虑完整的Apriori算法，伪代码如下：

```
IF 集合中项的个数大于 0:
    构建一个候选 k 项集列表
    确认每个项集都是频繁的
    保留频繁项集并构建候选 k + 1 项集列表
```

具体的Python实现如下：

```
def aprioriGen(Lk, k):
    retList = []
    lenLk = len(Lk)
    for i in range(lenLk):
        for j in range(i + 1, lenLk):
            L1 = list(Lk[i])[:k - 2]; L2 = list(Lk[j])[:k - 2]
            L1.sort(); L2.sort()
            if L1 == L2:
                retList.append(Lk[i] | Lk[j])
    return retList

def apriori(dataSet, minSupport = 0.5):
    C1 = createC1(dataSet)
    L1, supportData = scanD(dataSet, C1, minSupport)
    L = [L1]
    k = 2
    while (len(L[k - 2]) > 0):
```

```
        Ck = aprioriGen(L[k-2], k)          # Ck
        Lk, supK = scanD(dataSet, Ck, minSupport)
        supportData.update(supK)
        L.append(Lk)
        k += 1
    return L, supportData
```

其中，aprioriGen()函数用于创建候选集，其输入参数为一个频繁项集列表 Lk 和项集元素个数 k，例如若输入列表 1，2，3，并将 k 设为 2，则返回{1，2}、{1，3}、{2，3} 3 个项集。apriori()函数则用于执行具体的 Apriori 算法。该函数需要输入一个数据集，并给定最小支持度阈值（若没有，则默认为 0.5），执行算法后返回所有的频繁项集。

下面测试一下代码效果：

```
>>> L, suppData = apriori(dataSet)
>>> L
[[frozenset({5}), frozenset({2}), frozenset({3}), frozenset({1})], [frozenset({2, 3}),
frozenset({3, 5}), frozenset({2, 5}), frozenset({1, 3})], [frozenset({2, 3, 5})], []]
```

计算结果 L 中包含了所有的频繁项集，也可单独查看 k 项集：

```
>>> L[0]
[frozenset({5}), frozenset({2}), frozenset({3}), frozenset({1})]>>> L[1]

>>> L[2]
[frozenset({2, 3}), frozenset({3, 5}), frozenset({2, 5}), frozenset({1, 3})]

>>> L[3]
[frozenset({2, 3, 5})]
```

如果要修改最小支持度阈值，可将该值作为第二个参数传递给 apriori()函数：

```
>>> L, suppData = apriori(dataSet, minSupport = 0.7)
>>> L
[[frozenset({5}), frozenset({2}), frozenset({3})], [frozenset({2, 5})], []]
```

到此为止已经可以从原始事务集中挖掘出频繁项集了，接下来的工作是从中提取出关联规则。

```
def generateRules(L, supportData, minConf = 0.7):
    bigRuleList = []
```

```
    for i in range(1, len(L)):
        for freqSet in L[i]:
            H1 = [frozenset([item]) for item in freqSet]
            if (i > 1):
                rulesFromConseq(freqSet, H1, supportData, bigRuleList, minConf)
            else:
                calcConf(freqSet, H1, supportData, bigRuleList, minConf)      #调用函数
    return bigRuleList

def calcConf(freqSet, H, supportData, brl, minConf = 0.7):
    prunedH = []
    for conseq in H:
        conf = supportData[freqSet]/supportData[freqSet - conseq]
        if conf >= minConf:
            print (freqSet - conseq, '-->', conseq, 'conf:', conf)
            brl.append((freqSet - conseq, conseq, conf))
            prunedH.append(conseq)
    return prunedH

def rulesFromConseq(freqSet, H, supportData, brl, minConf = 0.7):
    m = len(H[0])
    if (len(freqSet) > (m + 1)):
        Hmp1 = aprioriGen(H, m + 1)
        Hmp1 = calcConf(freqSet, Hmp1, supportData, brl, minConf)
        if (len(Hmp1) > 1):
            rulesFromConseq(freqSet, Hmp1, supportData, brl, minConf)
```

在上述代码中 generateRules()是主函数，用于生成关联规则，需要 3 个参数，即频繁项集列表、包含频繁项集支持数据的字典、最小置信度阈值，其中前两个参数可从 apriori()函数的返回值中获取，最小置信度阈值若不给定，默认为 0.7。calcConf()函数用于计算置信度，并返回一个满足最小置信度要求的规则列表。rulesFromConseq()函数用于从最初的项集中生成更多的关联规则。

下面测试代码的运行效果：

```
>>> L, suppData = apriori(dataSet, minSupport = 0.5)
>>> rules = generateRules(L, suppData, minConf = 0.7)
frozenset({5}) --> frozenset({2}) conf: 1.0
frozenset({2}) --> frozenset({5}) conf: 1.0
frozenset({1}) --> frozenset({3}) conf: 1.0
[(frozenset({5}), frozenset({2}), 1.0), (frozenset({2}), frozenset({5}), 1.0),
        (frozenset({1}), frozenset({3}), 1.0)]
```

可以看到，这里生成了 3 条强关联规则：5→2、2→5、1→3，3 条规则的置信度均为 1，这意味着只要出现 5，则必然出现 2，只要出现 2，则必然出现 5，只要出现 1，则必然出现 3，可见关联规则的前导和后继在某些情况下可互换，在某些情况下却不行。

若将最小置信度阈值降低，例如设为 0.5，可获得更多的规则：

```
>>> L, suppData = apriori(dataSet, minSupport = 0.5)
>>> rules = generateRules(L, suppData, minConf = 0.5)
frozenset({3}) --> frozenset({2}) conf: 0.6666666666666666
frozenset({2}) --> frozenset({3}) conf: 0.6666666666666666
frozenset({5}) --> frozenset({3}) conf: 0.6666666666666666
frozenset({3}) --> frozenset({5}) conf: 0.6666666666666666
frozenset({5}) --> frozenset({2}) conf: 1.0
frozenset({2}) --> frozenset({5}) conf: 1.0
frozenset({3}) --> frozenset({1}) conf: 0.6666666666666666
frozenset({1}) --> frozenset({3}) conf: 1.0
frozenset({5}) --> frozenset({2, 3}) conf: 0.6666666666666666
frozenset({3}) --> frozenset({2, 5}) conf: 0.6666666666666666
frozenset({2}) --> frozenset({3, 5}) conf: 0.6666666666666666
[(frozenset({3}), frozenset({2}), 0.6666666666666666), (frozenset({2}), frozenset({3}),
0.6666666666666666), (frozenset({5}), frozenset({3}), 0.6666666666666666),
(frozenset({3}), frozenset({5}), 0.6666666666666666), (frozenset({5}), frozenset({2}),
1.0), (frozenset({2}), frozenset({5}), 1.0), (frozenset({3}), frozenset({1}),
0.6666666666666666), (frozenset({1}), frozenset({3}), 1.0), (frozenset({5}),
frozenset({2, 3}), 0.6666666666666666), (frozenset({3}), frozenset({2, 5}),
0.6666666666666666), (frozenset({2}), frozenset({3, 5}), 0.6666666666666666)]
```

当然，在现实情况中，即使是强关联规则也不一定能引起人们的兴趣，因为有可能产生一些显而易见的规则，需要后期进行人工筛选。一旦算法产生的规则过多，后期筛选的成本也会相应提高，因此最小置信度设为多少更为合理需要仔细考虑。

### 11.3.3 FP-growth 算法

#### 1. 算法简介

上一节学习的 Apriori 算法通过不断地构造候选集、筛选候选集挖掘出频繁项集，需要多次扫描原始数据，当原始数据较大时，磁盘 I/O 次数太多，效率比较低。FP-growth 算法是基于 Apriori 原理的，通过将数据集存储在 FP(Frequent Pattern)树上发现频繁项集，但不能发现数据之间的关联规则。FP-growth 算法只需要对数

据库进行两次扫描，而 Apriori 算法在求每个潜在的频繁项集时都需要扫描一次数据集，因此 FP-growth 算法相对 Apriori 算法来说更为高效。其中，算法发现频繁项集的过程如下：

(1) 构建 FP 树。

(2) 从 FP 树中挖掘频繁项集。

FP 树通过逐个读入事务，并把事务映射到 FP 树中的一条路径来构造。由于不同的事务可能会有若干个相同的项，所以它们的路径可能部分重叠。路径相互重叠越多，使用 FP 树结构获得的压缩效果越好；如果 FP 树足够小，能够存放在内存中，就可以直接从这个内存中的结构提取频繁项集，而不必重复地扫描存放在硬盘上的数据。

一棵 FP 树如图 11.6 所示。

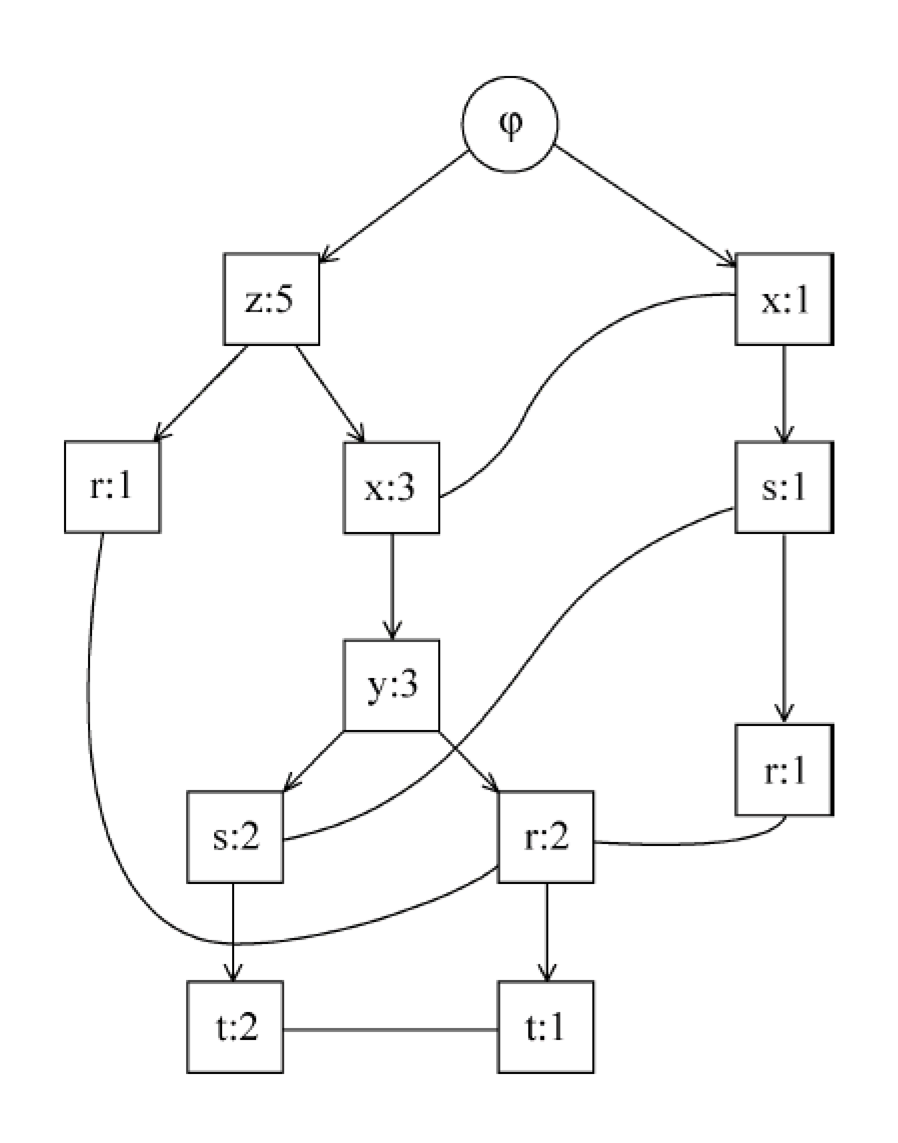



**图 11.6　典型 FP 树示例**

FP 树的根结点用 φ 表示，其余结点包括一个数据项和该数据项在本路径上的支持度；每条路径都是一条训练数据中满足最小支持度的数据项集，路径用箭头线条表示；FP 树还将所有相同项连接成链表，用线条表示。

为了快速访问树中的相同项，还需要维护一个连接具有相同项的结点的指针列表(headTable)，每个列表元素包括数据项、该项的全局最小支持度、指向 FP 树中该项链表的表头的指针，如图 11.7 所示。

FP-growth 算法需要对原始训练集扫描两遍以构建 FP 树。第一次扫描，过滤掉

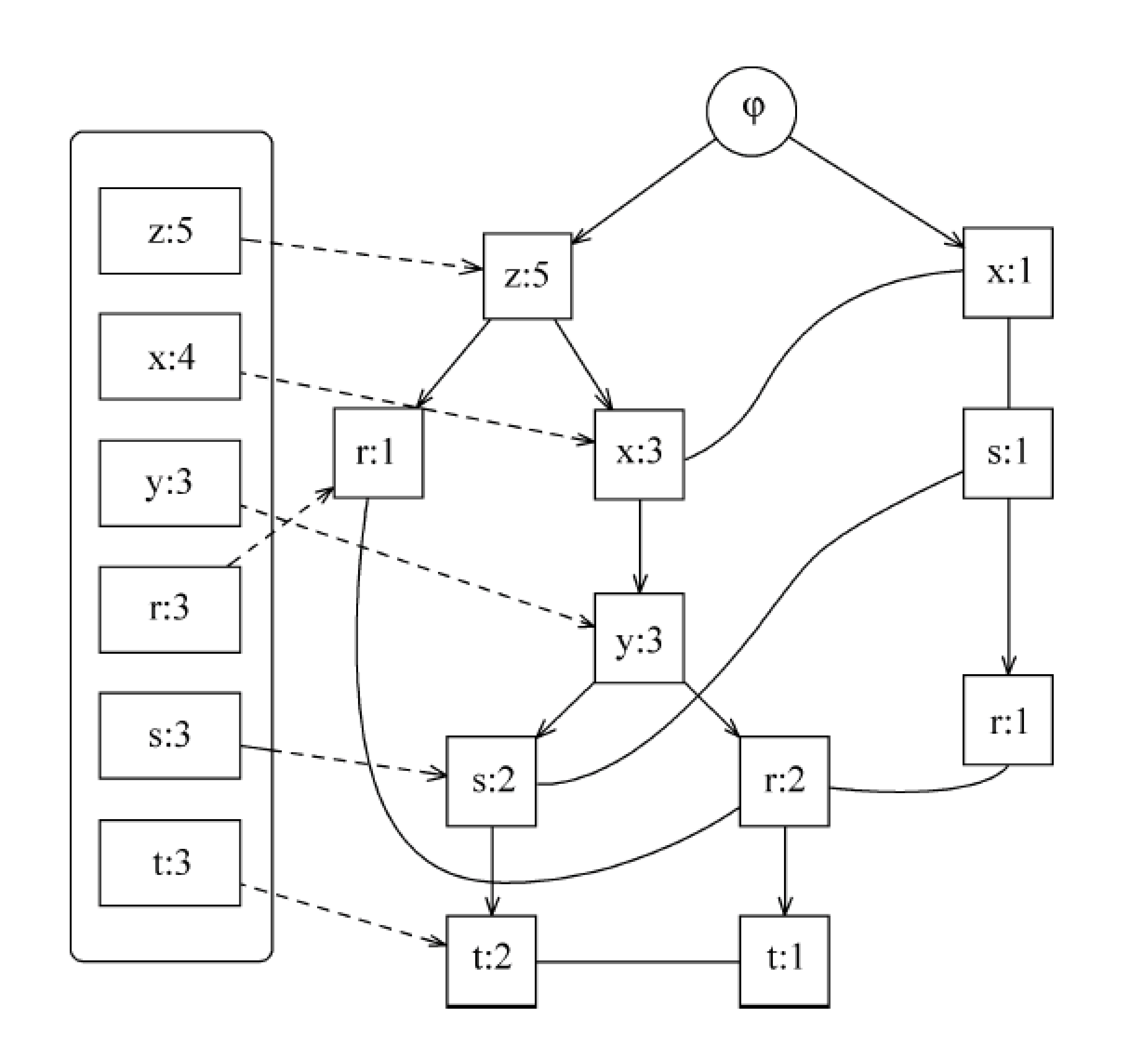



图 11.7 具有结点指针列表的 FP 树

所有不满足最小支持度的项；对于满足最小支持度的项，按照全局最小支持度排序，在此基础上，为了处理方便，也可以按照项的关键字再次排序。

现在假设有如表 11.2 所示的数据。

**表 11.2 原始数据集**

| 事务 ID | 事务中的项 | 事务 ID | 事务中的项 |
|---|---|---|---|
| 1 | r，z，h，j，p | 4 | r，x，n，o，s |
| 2 | z，y，x，w，v，u，t，s | 5 | y，r，x，z，q，t，p |
| 3 | z | 6 | y，z，x，e，q，s，t，m |

首先第一次遍历整个数据集，统计每个元素项的出现频率，去掉不满足最小支持度(这里假设为 3)的元素项，处理的结果如表 11.3 所示。

**表 11.3 移除非频繁项后的数据**

| 事务 ID | 事务中的项 | 过滤及重新排序后的项 |
|---|---|---|
| 1 | r，z，h，j，p | z，r |
| 2 | z，y，x，w，v，u，t，s | z，x，y，s，t |
| 3 | z | z |
| 4 | r，x，n，o，s | x，s，r |
| 5 | y，r，x，z，q，t，p | z，x，y，r，t |
| 6 | y，z，x，e，q，s，t，m | z，x，y，s，t |

注意到这里由于将最小支持度设为3，所以 e、j、m、p、q、v、u、w、o 等项被去除掉，仅剩下 z、x、y、r、s、t 等项。

接下来进行第二次扫描，同时构建 FP 树。参与扫描的是过滤后的数据，如果某个数据项是第一次遇到，则创建该结点，并在 headTable 中添加一个指向该结点的指针；否则按路径找到该项对应的结点，修改结点信息。具体过程如图 11.8～图 11.13 所示。

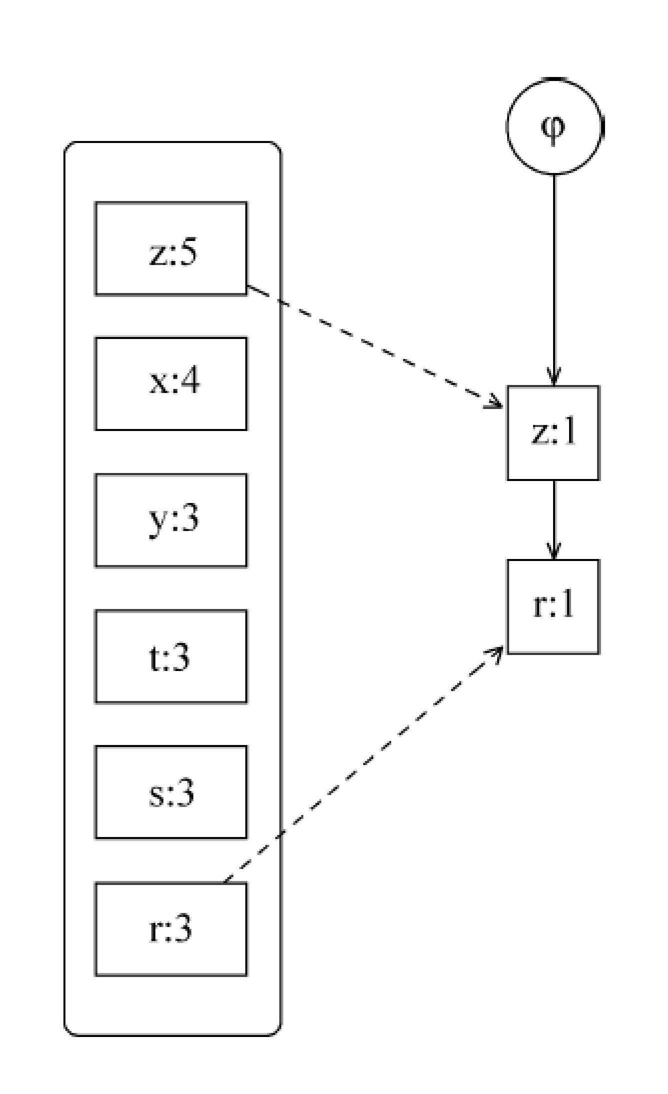



图 11.8　扫描事务 1 后的 FP 树

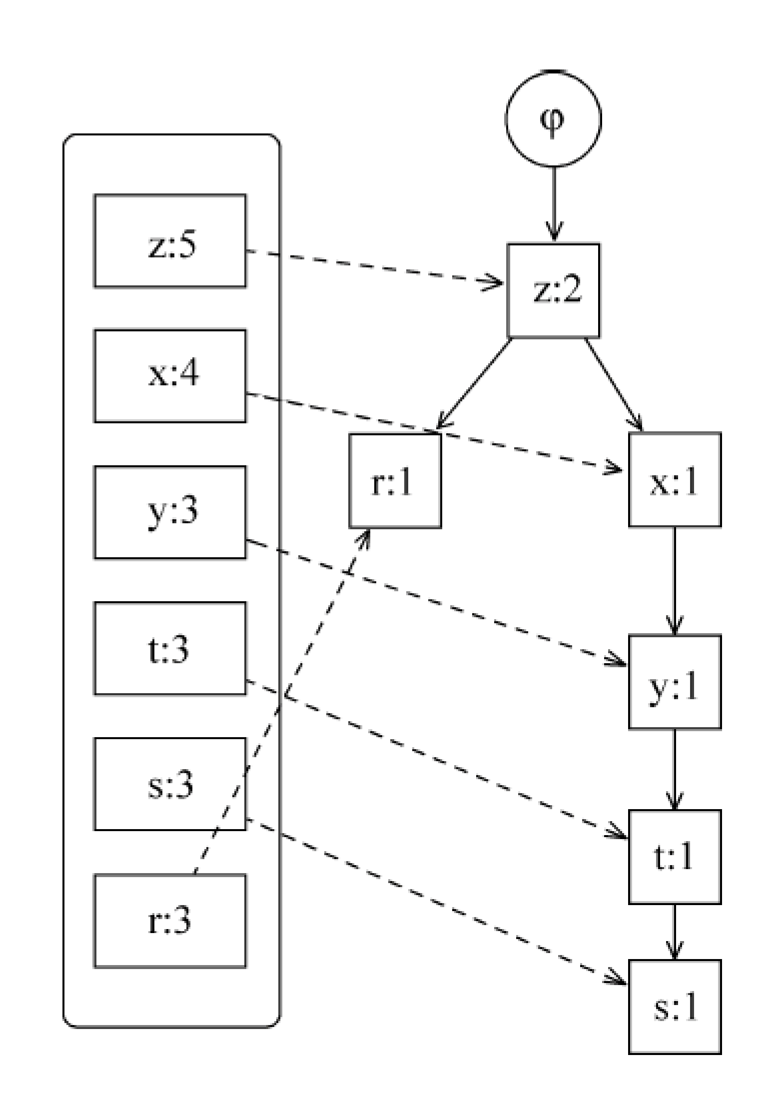



图 11.9　扫描事务 2 后的 FP 树

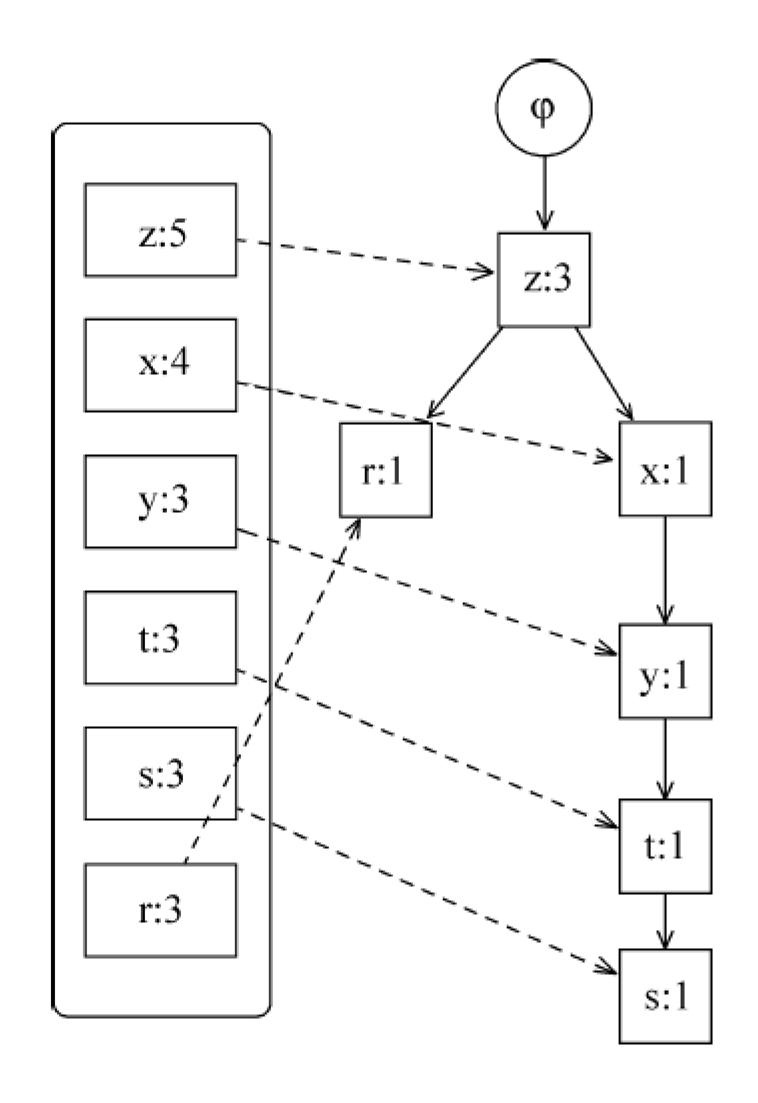



图 11.10　扫描事务 3 后的 FP 树

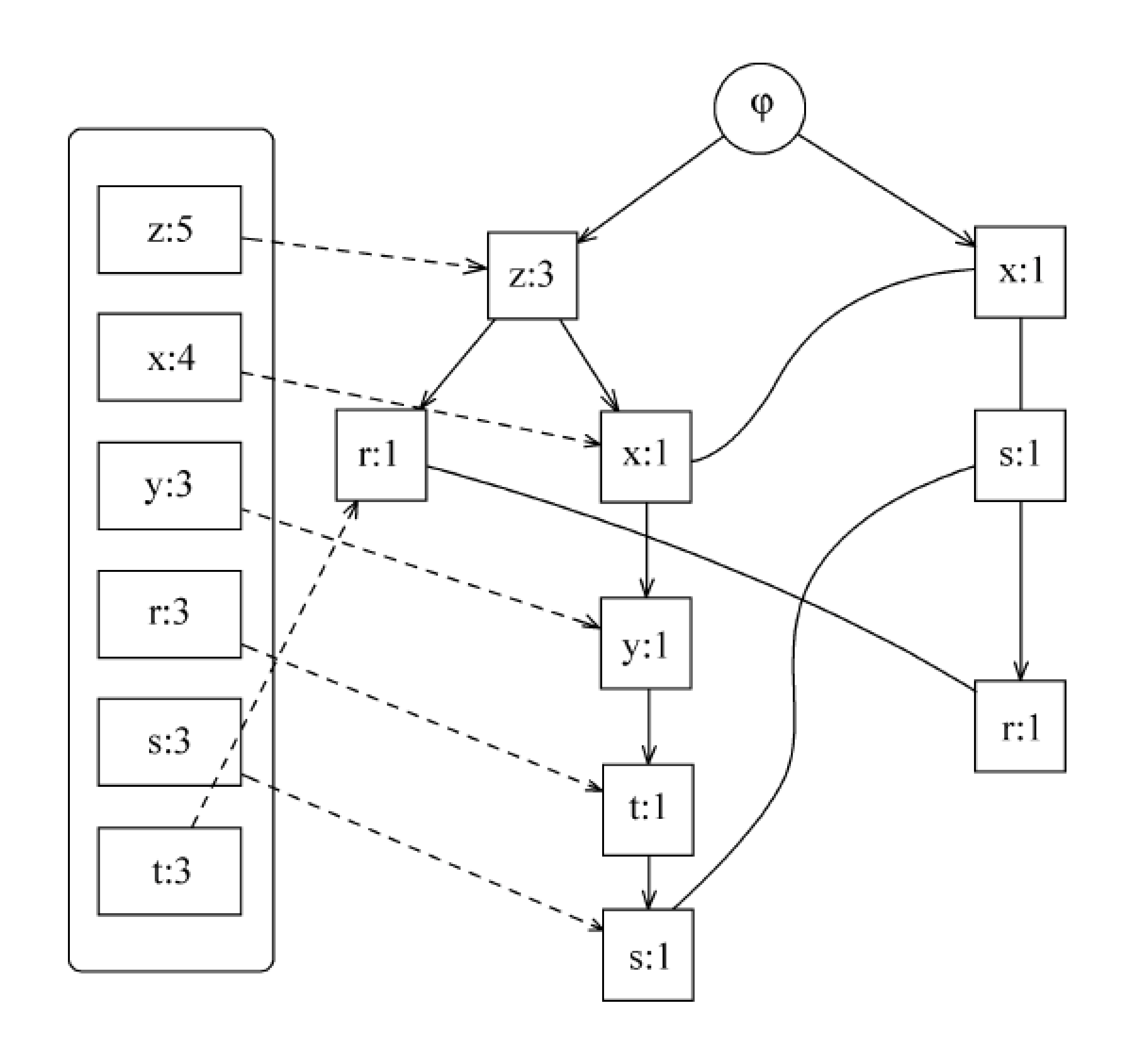



图 11.11 扫描事务 4 后的 FP 树

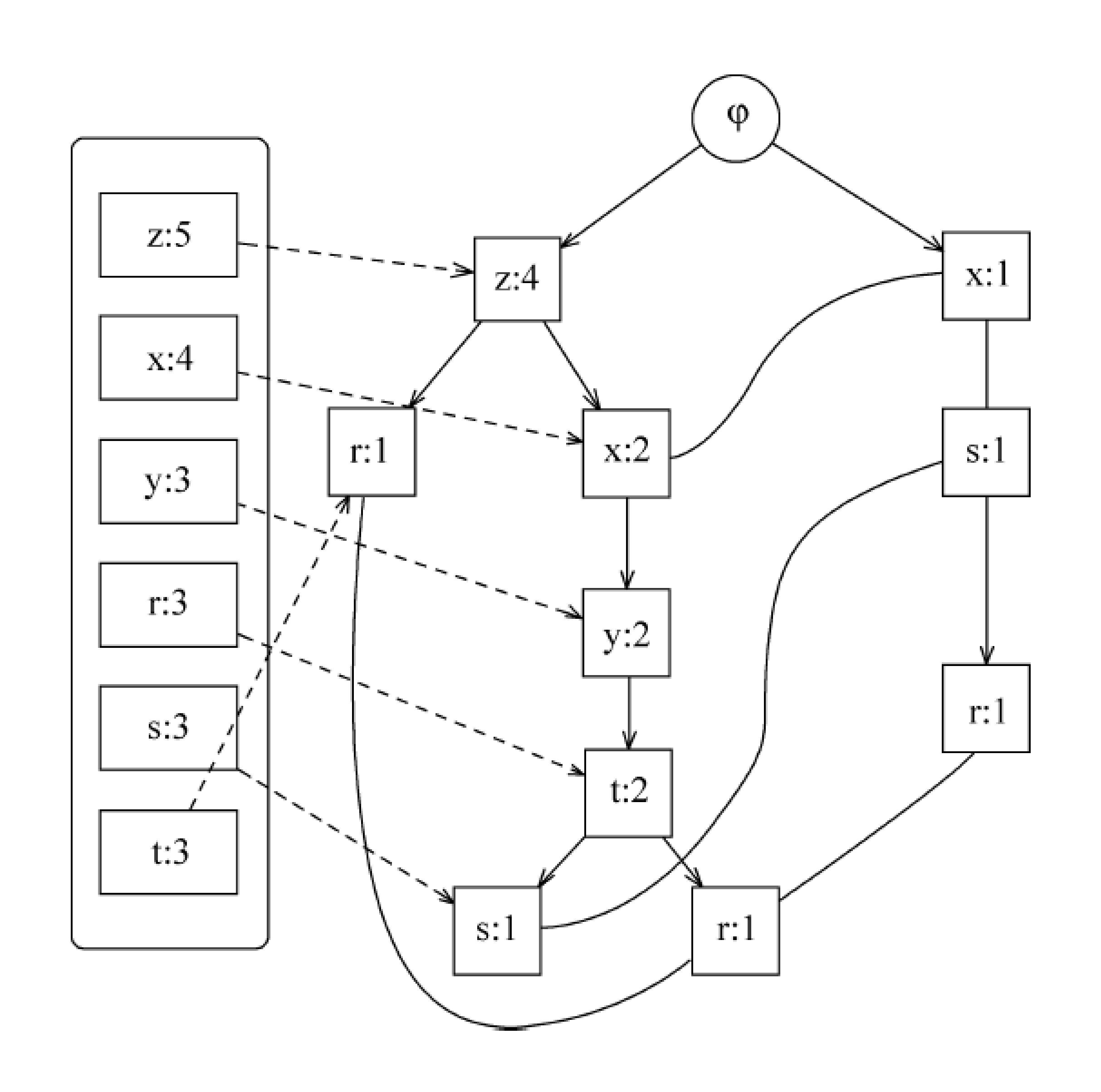



图 11.12 扫描事务 5 后的 FP 树

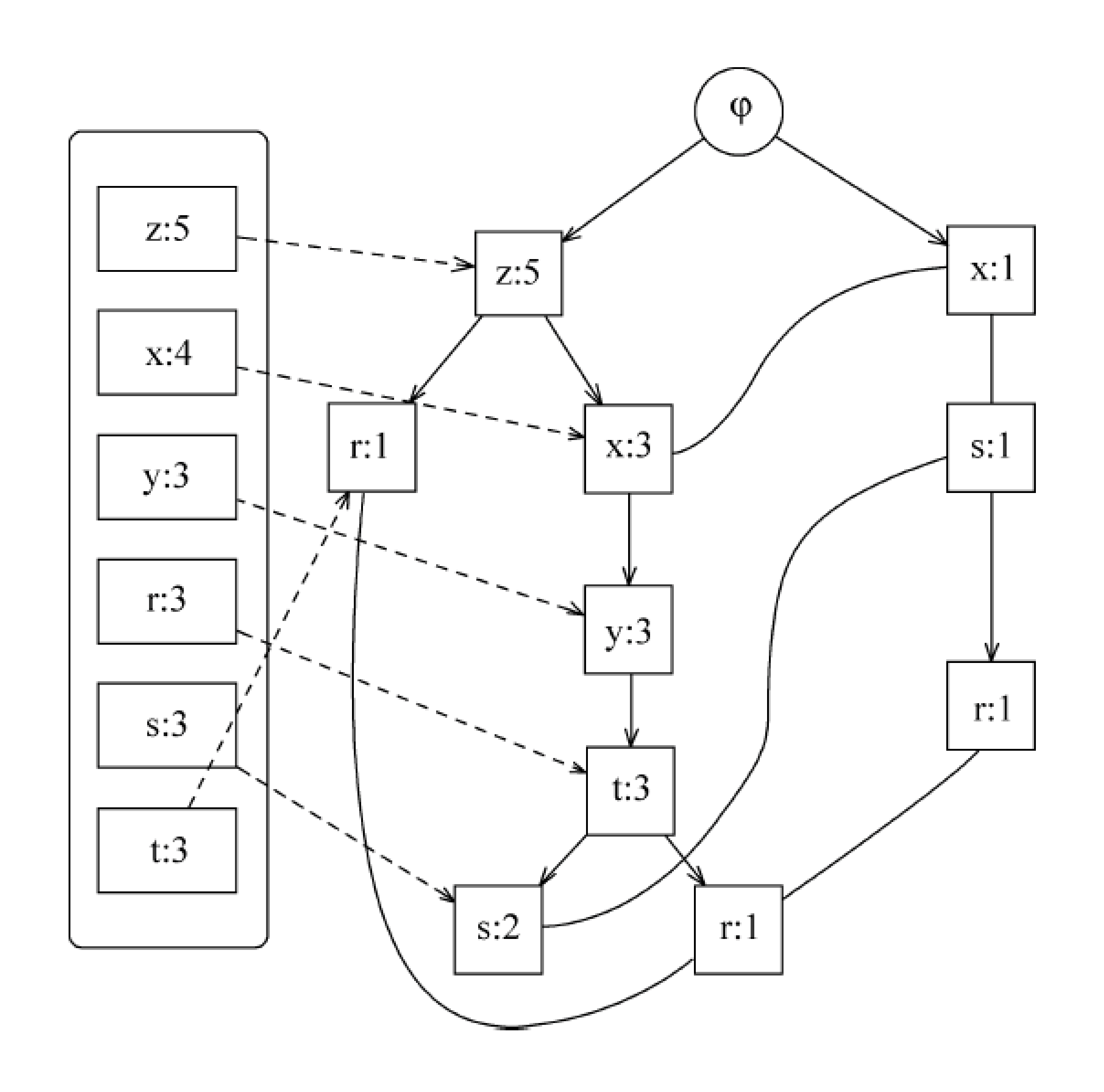



图 11.13　扫描事务 6 后的 FP 树

### 2. FP-growth 算法的 Python 实现

为了构建 FP 树，需要构建一个 FP 树的数据结构：

```
class treeNode:
    def __init__(self, nameValue, numOccur, parentNode):
        self.name = nameValue
        self.count = numOccur
        self.nodeLink = None
        self.parent = parentNode
        self.children = {}

    def inc(self, numOccur):
        self.count  += numOccur

    def disp(self, ind = 1):
        print('   ' * ind, self.name, '', self.count)
        for child in self.children.values():
            child.disp(ind + 1)
```

该类中包含了用于存放结点名字的变量和一个计数值，还用了父变量 parent 来指向当前结点的父结点。此外，类中还包含一个空字典变量，用于存放结点的子结点。inc()方法用于增加 count 的数值。disp()函数以文本形式显示 FP 树。

下面测试一下上述代码：

```
>>> rootNode = treeNode('a', 9, None)
>>> rootNode.children['b'] = treeNode('b', 13, None)
>>> rootNode.disp()
   a   9
   b   13
```

下面是构建 FP 树的函数：

```
def createTree(dataSet, minSup = 1):
    headerTable = {}
    for trans in dataSet:
        for item in trans:
            headerTable[item] = headerTable.get(item, 0) + dataSet[trans]
    for k in list(headerTable):
        if headerTable[k] < minSup:
            del (headerTable[k])
    freqItemSet = set(headerTable.keys())
    if len(freqItemSet) == 0:
        return None, None
    for k in headerTable:
        headerTable[k] = [headerTable[k], None]
    retTree = treeNode('Null Set', 1, None)
    for tranSet, count in dataSet.items():
        localD = {}
        for item in tranSet:
            if item in freqItemSet:
                localD[item] = headerTable[item][0]

        if len(localD) > 0:
            orderedItems = [v[0] for v in sorted(localD.items(), key = lambda p: p[1],
reverse = True)]
            updateTree(orderedItems, retTree, headerTable, count)
    return retTree, headerTable


def updateTree(items, inTree, headerTable, count):
    if items[0] in inTree.children:
        inTree.children[items[0]].inc(count)
    else:
        inTree.children[items[0]] = treeNode(items[0], count, inTree)
        if headerTable[items[0]][1] == None:
            headerTable[items[0]][1] = inTree.children[items[0]]
        else:
            updateHeader(headerTable[items[0]][1], inTree.children[items[0]])
    if len(items) > 1:
        updateTree(items[1::], inTree.children[items[0]], headerTable, count)
```

```
def updateHeader(nodeToTest, targetNode):
    while (nodeToTest.nodeLink != None):
        nodeToTest = nodeToTest.nodeLink
    nodeToTest.nodeLink = targetNode
```

第一个函数 createTree()用于构建 FP 树，需要输入数据集及最小支持度作为参数。树的构建过程需要遍历数据集两次，第一次遍历统计每个元素项出现的频度，第二次遍历频繁项集，并调用 updateTree()方法对 FP 树进行填充更新。

为了查看 FP 树的构建效果，这里手工创建一些实例数据：

```
def loadSimpDat():
    simpDat = [['r', 'z', 'h', 'j', 'p'],
               ['z', 'y', 'x', 'w', 'v', 'u', 't', 's'],
               ['z'],
               ['r', 'x', 'n', 'o', 's'],
               ['y', 'r', 'x', 'z', 'q', 't', 'p'],
               ['y', 'z', 'x', 'e', 'q', 's', 't', 'm']]
    return simpDat

def createInitSet(dataSet):
    retDict = {}
    for trans in dataSet:
        retDict[frozenset(trans)] = 1
    return retDict
```

下面查看代码的运行效果：

```
>>> simpDat = loadSimpDat();
>>> initSet = createInitSet(simpDat)
>>> myFPtree, myHeaderTab = createTree(initSet, 3)
>>> myFPtree.disp()
    Null Set   1
     z   5
       r   1
       x   3
         s   2
           t   2
             y   2
         r   1
           t   1
             y   1
     x   1
       s   1
         r   1
```

结果中给出了各元素项及其频度,每个缩进表示其所处的树的深度。

有了 FP 树之后就可以开始从中提取频繁项集了。首先从 1 项集开始,逐渐构建更大的项集。在构建时完全基于 FP 树,而不需要读取原始事务集。整个提取步骤包括:

(1) 从 FP 树中获得条件模式基。

(2) 利用条件模式基构建一个条件 FP 树。

(3) 重复步骤(1)和(2),直到树包含一个元素项为止。

首先从 FP 树头指针表中的单个频繁元素项开始。对于每一个元素项,获得其对应的条件模式基,单个元素项的条件模式基也就是元素项的关键字。条件模式基是以所查找元素项为结尾的路径集合。每一条路径其实都是一条前缀路径。简而言之,一条前缀路径是介于所查找元素项与树根结点之间的所有内容。

下列代码给出了前缀路径发现的过程:

```
def ascendTree(leafNode, prefixPath):
    if leafNode.parent != None:
        prefixPath.append(leafNode.name)
        ascendTree(leafNode.parent, prefixPath)


def findPrefixPath(basePat, treeNode):
    condPats = {}
    while treeNode != None:
        prefixPath = []
        ascendTree(treeNode, prefixPath)
        if len(prefixPath) > 1:
            condPats[frozenset(prefixPath[1:])] = treeNode.count
        treeNode = treeNode.nodeLink
    return condPats
```

上述代码通过访问树中所有包含给定元素项的结点为给定元素项生成一个条件模式基。

对于每一个频繁项集,都需要构建一棵条件 FP 树。

```
def mineTree(inTree, headerTable, minSup, preFix, freqItemList):
    bigL = [v[0] for v in sorted(headerTable.items(), key = lambda p: str(p[1]))]
    for basePat in bigL:
        newFreqSet = preFix.copy()
```

```
        newFreqSet.add(basePat)
        freqItemList.append(newFreqSet)
        condPattBases = findPrefixPath(basePat, headerTable[basePat][1])
        myCondTree, myHead = createTree(condPattBases, minSup)
        if myHead != None:
            print ('conditional tree for: ',newFreqSet)
            myCondTree.disp(1)
            mineTree(myCondTree, myHead, minSup, newFreqSet, freqItemList)
```

该函数通过递归去查找频繁项集，直到树中没有元素项为止。

## 本章小结

(1) 本章介绍了聚类问题，并给出了两个常用算法 K-Means 和 DBSCAN。这两个算法各有优劣，需要针对数据的具体情况斟酌使用。

(2) 本章介绍了关联分析问题，并给出了两个常用算法 Apriori 和 FP-growth。

(3) 本章的 4 个算法均给出了基于 Python 的具体实现和应用。

## 习题

1. 在 K-Means 算法中，比较取不同 K 值时的聚类效果。
2. K-Means 和 DBSCAN 算法各有何优劣？
3. 为什么说挖掘关联规则的核心是挖掘频繁项集？
4. 如何理解 FP-growth 算法相比 Apriori 来说更为高效？

# 第 12 章

# Python地理空间分析

**本章学习目标：**

- 理解地理空间分析的基本概念
- 了解常用的地理空间数据及其组织、结构
- 熟练掌握常用的 Python 地理空间分析工具
- 熟练掌握使用 Python 进行针对地理信息系统的地理空间分析
- 熟练掌握使用 Python 进行针对遥感的地理空间分析

## 12.1 地理空间分析简介

### 12.1.1 地理空间分析的基本概念

随着现代科学技术(尤其是计算机技术)引入地图学和地理学，地理信息系统开始孕育、发展，以数字形式存在于计算机中的地图向人们展示了更为广阔的应用领域。利用计算机分析地图、获取信息、支持空间决策成为地理信息系统的重要研究内容，“空间分析”这个词也就成为这一领域的一个专门术语。

空间分析主要通过空间数据和空间模型的联合分析来挖掘空间目标的潜在信息，而这些空间目标的基本信息无非是其空间位置、分布、形态、距离、方位、拓扑关系等，其中距离、方位、拓扑关系组成了空间目标的空间关系，它是地理实体之间的空间特性，可以作为数据组织、查询、分析和推理的基础。通过将地理空间目标划分为点、线、面不同的类型，可以获得这些不同类型目标的形态结构。将空间目标的空间数据和属性数据结合起来，可以进行许多特定任务的空间计算与分析。

任何信息都含有空间、时间、属性特征。例如水文走向含属性和时间特征，疾病传播含时间、空间和属性特征，而河道演变则反映了空间形态特征随时间变化的性质。国内外许多学者都对空间分析进行研究，但是对空间分析下定义是比较困难的，目前尚无一个统一的定义，不同的应用领域给出不同的含义，它们的侧重点各不相同，但是都从不同方面对空间分析的内涵进行了阐释，或侧重于地理学（地学），或侧重于测绘学（地图学），或侧重于几何图形分析，或侧重于地学统计与建模。综合这些学者的研究成果，GIS空间分析是使用几何分析、统计分析、数学建模、地理计算等方法对地理空间中目标的空间关系进行描述、分析、建模，并进一步为空间决策支持提供服务的技术。

### 12.1.2　地理空间分析与 Python

现代的地理空间分析可以通过商业或开源的地理空间应用软件轻松完成，但是有时候使用编程语言进行地理空间分析也有必要，例如：

（1）希望完全控制底层的算法、数据和执行过程。

（2）希望用最小的代价在一个大而全的地理空间框架中实现重复任务的自动化。

（3）希望创建一个程序方便共享。

（4）希望深入学习地理空间分析，而不只是击一下鼠标按键。

地理空间行业正逐渐脱离曾经的那种需要分析团队通过昂贵的桌面软件生产地理空间产品的传统工作模式。当前地理空间分析趋向于通过云模式进行自动化过程处理。终端用户软件趋向于解决特定任务的工具，而且很多还支持移动设备访问。地理空间概念和数据的知识与个性化地理空间分析过程一样，是人们将来从事地理空间工作的基础。

Python 语言以“多面手”而闻名，它可以是设计良好的面向对象编程语言，也可以是过程式脚本语言，甚至还可以是函数式编程语言。但总的来说面向对象是Python 天然的也是最大的优势。地理空间分析和面向对象编程是天生的一对。在大部分面向对象编程项目中对象都是抽象的概念，例如数据库连接在现实生活中是无法找到参照物的，但是在地理空间分析领域，被建模的对象都可以和现实生活对应。地理空间分析研究的范围就是地球上的一切，例如植被、建筑物、河流、人以及其他对象共同组成了地理空间系统。

## 12.2 地理空间数据

### 12.2.1 数据格式概览

对于地理空间分析来说，最常见到的主要是如下数据格式：

- 电子表格、逗号分隔文件(CSV 文件)。
- 地理标记照片。
- 轻量级的二进制点、线和多边形。
- GB 级别的卫星和航空影像。
- 高程数据，例如网格、点云和基于整数的影像。
- XML 文件。
- JSON 文件。
- 数据库。
- Web 服务。

每种格式都会在访问和处理方面存在特有的问题，所以在进行数据分析时通常不得不预先处理它们。例如也许需要从一张大的卫星影像中剪裁或者提取感兴趣的区域，或者想减少数据集中点的数量只留下符合特定标准的部分，这通常被称为预处理。另外，地理空间数据有其特有的预处理过程，即投影。

在本章中有两个概念经常被提及，即矢量数据和栅格数据，它们是大部分地理空间数据集分类的默认方式。矢量数据可以是任何包含点、线、多边形等地理位置数据的文件；栅格数据可以是任何以行列式网格存储数据的文件，栅格数据还包括所有

图片格式的文件。

### 12.2.2 数据特征

地理空间数据有一些共同的特征，用户了解这些特征有利于通过空间数据的共性了解那些不熟悉的数据格式。现有的数据格式的结构通常是由它的用途决定的，某些数据存储和压缩很高效，某些数据访问性好，某些数据轻量级、易读，某些数据兼容性好。当然，易用性对于地理分析来说是非常重要的，简单的数据格式对于整合和分析来说能提供最大的支持。

地理空间分析是信息处理技术在地理学上的应用，因此地理空间数据有两个最重要的组成部分，即地理位置和主题信息。此外，空间数据的另外一个共性是空间索引。

(1) 地理位置：地理信息是由地球上的一个点和与之相关的信息所构成的。这个位置通常是球面上的一个坐标点(用经纬度来表示)，或者是将其映射到一个平面空间上的一点(用 x，y 坐标表示)。

(2) 主题信息：主题信息涵盖的范围广泛，它可以是组成地面可视影像图片的像素，也可以是一张处理多光谱的图片，还可以是某种数据库中行和列的每种地理化特征。

(3) 空间索引：地理空间数据集文件通常都非常大，动辄几百 MB 到几 GB 大小，因此在处理和执行分析中多次访问大型文件时效率非常低下。空间索引能够创建一个向导，让软件无须扫描数据集中的每一行记录而快速定位查询结果。常用的空间索引算法为四叉树索引和 R 树索引。

### 12.2.3 矢量数据

矢量数据是存储空间信息最有效的一种方式，一般来讲计算机通过矢量存储和处理地理信息所需的资源要远少于栅格数据。例如，如果要表达一条直线道路，矢量数据只需要存储道路起点和终点的坐标即可，而栅格数据则要存储这条直线道路上的每一个点(至于具体需要多少个点来存储这条道路，取决于该栅格数据的分辨率)。

与计算机图形学中的矢量概念不同的是，地理空间矢量数据可以有正/负值(取决于地理原点)，并且地理空间矢量数据还包含一些与几何对象相关的其他信息，例

如表达房屋坐落的矢量点除了存储房屋位置之外，还会存储该房屋的类型、所有人、建造时间等；表达道路的矢量线除了存储道路位置之外，还会存储道路级别、道路名称、是否单行等信息。

目前可用的矢量数据格式超过 200 种，非常多，既有开源的矢量格式，也有商业软件内置的矢量格式。矢量数据文件可以是二进制存储，也可以是纯文本文件存储。其中，纯文本矢量数据文件包括 CSV、GeoJSON 和 XML 等，该类型文件对于计算机和人类来说都是可读的，但是文件大小比二进制文件大很多。二进制矢量数据文件中最流行的是 Shapefile。

Shapefile 最早由 ESRI 公司提出，并在 1998 年将其标准作为一种开放规范发布。该格式因其规范开放、高效、简单而逐渐成为事实上的地理信息系统标准。今天几乎所有跟地理空间相关的软件和开发工具都提供了对 Shapefile 文件的支持。在 Python 中可以通过 OGR 库模块实现对 Shapefile 的解析和操作。

Shapefile 格式的特点之一是它由多个文件组成，其中 3 种文件是必需的，另外有十几种可选扩展文件。表 12.1 描述了部分 Shapefile 文件格式信息。

**表 12.1　Shapefile 文件格式信息**

| 扩　展　名 | 用　　途 | 备　　注 |
|---|---|---|
| .shp | 用于存储要素几何的主文件，其中包括几何图形 | 必要文件 |
| .shx | 形状索引文件，适当尺寸的几何元素索引信息可以加快访问速度 | 必要文件。这个文件必须和.shp 文件配合使用，否则没有意义 |
| .dbf | 数据库文件，其中包括几何元素的属性信息 | 必要文件 |
| .sbn | 空间.bin 文件，Shapefile 文件的索引文件 | 包含一个特征的边框，映射了一个 256×256 的整数网格 |
| .sbx | .sbn 文件的索引记录文件 | 打开一个文件，只写。如果该文件存在，则覆盖该文件；如果该文件不存在，则在该路径下创建一个新的文件，用于写入 |
| .prj | 以 WKT 格式存储的地图投影信息 | 很常见的文件，常用于 GIS 软件中的地图实时投影 |

对于一个 Shapefile 文件来说，都是由若干上述文件共同构成的，这意味着每个 Shapefile 文件都会具有同名的几个文件。如果需要对一个 Shapefile 文件重命名，那么需要确保将上述所有文件改成同一个名字。在各种 GIS 软件中会将这些数据集当成同一个文件看待。

### 12.2.4　栅格数据

在地理信息系统领域更多地使用术语“栅格”来指代影像数据。栅格数据由若干行或列的单元或像素(术语称为像元)所构成,每个单元表达为一个数值,该数值可以代表计算机的显示亮度,也可以是真实世界的物理量(例如辐射强度或光谱反射率)。用户可以将栅格数据理解为计算机中的数字图片,特殊之处在于栅格图像反映的通常是地理信息,更多时候是航天卫星拍摄下来的数字影像。

栅格数据集可能包含多个波段,这意味着可以同时收集同一区域中不同波长的光谱信息,常见的航天卫星能够同时记录 3～7 个波段(例如红光、蓝光、绿光、近红外等),有的航天卫星能够记录同一区域中的数百个光谱波段(这被称为高光谱卫星)。

栅格数据也包含多达数百种格式,其中一些常见格式为 TIFF、IMG、JPEG 等。TIFF(The Tagged Image File Format,标记化图片文件格式)是地理信息领域最常用的栅格格式,其灵活的标记系统允许用户在一个单独文件中存储任何类型的数据,可以包括概要图、多波段、整型高程数据、元数据等文件格式。GeoTIFF 扩展定义了地理空间数据的存储,常见的 TIFF 格式文件的扩展名为.tiff、.tif、.gtif。

## 12.3　Python 地理空间分析工具

### 12.3.1　GeoJSON

GeoJSON 是一种对各种地理数据结构进行编码的格式,基于 JavaScript 对象表示法的地理空间信息数据交换格式。GeoJSON 对象可以表示几何、特征或者特征集合。GeoJSON 支持点、线、面、多点、多线、多面和几何集合等几何类型。GeoJSON 里的特征包含一个几何对象和其他属性,特征集合表示一系列特征。下面是包含一个点对象的 GeoJSON 文档示例:

```
{
    "type": "Feature",
    "id": "OpenLayers.Feature.Vector_314",
    "properties": {},
    "geometry" {
```

```
            "type": "Point",
            "coordinates": [
                120.3154,
                30.1324
            ]
        },
        "crs": {
            "type": "name",
            "properties": {
              "name": "urn:ogc:def:crs:OGC:1.3:CRS84"
            }
        }
    }
```

为处理这样的 GeoJSON 数据，首先可以将此文档转换成一个字符串：

```
>>> jsdata = """{ "type": "Feature", "id": "OpenLayers.Feature.Vector_314",
     "properties": {}, "geometry": {"type": "Point", "coordinates": [
     120.3154, 30.1324 ]}, "crs": { "type": "name", "properties": { "name":
     "urn:ogc:def:crs:OGC:1.3:CRS84"}}}
```

GeoJSON 看上去就像是 Python 的字典和 List 对象的嵌套组合，因此需要使用 Python 的 eval()方法将其转换为 Python 代码。

```
>>> point = eval(jsondata)
>>> point["geometry"]
{'type': 'Point', 'coordinates': [120.3154, 30.1324]}
```

此外，在 Python 标准库中提供了专门的 json 模块来达到相同的目的，同样可以将上述字符串转换为 Python 代码：

```
>>> import json
>>> json.loads(jsdata)
```

反过来，用户也可以使用 json 模块中的 dumps()函数将 Python 数据结构转换为 JSON 格式的字符串：

```
>>> pydata = json.loads(jsdata)
>>> json.dumps(pydata)
```

虽然使用 json 模块已经可以很好地读写 GeoJSON 数据，但事实上还有更好的

解决方案——geojson 模块。该模块对 GeoJSON 格式数据提供了更方便的功能。例如，若要创建一个点要素并将其转换为 GeoJSON 格式，可以使用如下代码：

```
>>> import geojson
>>> p = geojson.Point([-92, 37])
>>> geojs = geojson.dumps(p)
>>> geojs
'{"type": "Point", "coordinates": [-92, 37]}'
```

### 12.3.2　GDAL 和 OGR

GDAL(Geospatial Data Abstraction Library)是一个目前最流行的在 X/MIT 许可协议下的开源栅格空间数据读写和转换库，大多数地理信息商业软件或开发包都在内部内置有该库以提供对栅格数据的全面支持。它利用抽象数据模型来表达所支持的各种文件格式。OGR 曾经是 GDAL 项目的一个分支，功能与 GDAL 类似，只不过它提供对矢量数据的支持。从 GDAL 2.0 开始，GDAL 和 OGR 已经被集成起来共同提供服务，所以通常把二者并称为 GDAL/OGR。

GDAL/OGR 使用统一的栅格/矢量抽象数据模型来表达它支持的所有空间数据。目前它提供的栅格数据格式驱动达到惊人的 154 种，提供的矢量数据格式驱动也达到 93 种。关于该项目的说明、下载及其支持的数据格式可在其官方网站查看，网址为"http://www.gdal.org/"。

GDAL/OGR 本身是使用 C/C++语言实现的，但是它也对其他编程语言例如 Python、Ruby、VB、Java、C# 提供了接口支持。其中，Python 的 GDAL/OGR 库可在"https://pypi.org/project/GDAL"网站找到。若用户要安装，可使用 pip 工具执行如下语句：

```
>>> pip install gdal
```

在安装后为了使用该库，需要将其导入：

```
>>> from osgeo import gdal
```

### 12.3.3　PyShp

PyShp 是 Python Shapefile Library 的简称，它使用纯 Python 语言来对 ESRI

Shapefile 格式文件提供读写支持，并对 Python 2 和 Python 3 同时提供兼容。前面提到，Shapefile 格式是当前最流行、最重要的地理信息系统矢量数据格式，因此掌握对 Shapefile 文件格式的读写是操作地理空间数据的基础。本书对 Shapefile 的操作主要通过 PyShp 库来完成。

截至本书出版 PyShp 的最新版本为 1.2.12，对于该项目的相关说明可在其官方网站找到(https://pypi.org/project/pyshp)，或者到其 GitHub 项目主页浏览相关代码。

PyShp 的安装文件可到其官网下载，或使用如下 pip 语句进行安装：

```
>>> pip install pyshp
```

### 12.3.4 PIL

PIL 库原本是用来处理遥感影像的，不过目前在 Python 中一般用于图像编辑。该库本身使用 C 语言实现，并专门针对 Python 做了一些优化。PIL 只支持 Python 2，对于 Python 3 来说，建议使用升级版本 Pillow，用户可通过官网“https://pypi.org/project/Pillow”下载并获取其更多相关信息。下面的代码实现了 Pillow 的常用操作：

```
>>> try:
>>>     import Image
>>>     import ImageDraw
>>> except:
>>>     from PIL import Image
>>>     from PIL import ImageDraw
>>> import shapefile
>>> r = shapefile.Reader("test.shp")
>>> xdist = r.bbox[2] - r.bbox[0]
>>> ydist = r.bbox[3] - r.bbox[1]
>>> iwidth = 400
>>> iheight = 600
>>> xratio = iwidth/xdist
>>> yratio = iheight/ydist
>>> pixels = []
>>> for x,y in r.shapes()[0].points:
...     px = int(iwidth - ((r.bbox[2] - x) * xratio))
...     py = int((r.bbox[3] - y) * yratio)
...     pixels.append((px, py))
...
>>> img = Image.new("RGB", (iwidth, iheight), "white")
>>> draw = ImageDraw.Draw(img)
>>> draw.polygon(pixels, outline="rgb(203, 196, 190)", fill("rgb(198, 204, 189)")
>>> img.save("test.png")
```

### 12.3.5　GeoPandas

Pandas 是一款高性能的 Python 数据分析库，它可以处理海量数据表格、矩阵以及无标记的统计数据。GeoPandas 是由多个地理空间处理库共同构建的 Pandas 地理空间扩展，旨在为 Python 处理空间数据提供更方便的操作。该库可以在其官网"https://pypi.org/project/geopandas"下载并找到相关信息。下面的代码将会打开一个 Shapefile 文件，将其转换成 GeoJSON 格式，并使用 matplotlib 库创建一张地图，可以看到整个操作极其整洁且方便。

```
>>> import geopandas
>>> import matplotlib.pyplot as plt
>>> gdf = geopandas.GeoDataFrame
>>> test_file = gdf.from_file("test.shp")
>>> test_file.plot()
>>> plt.show()
```

## 12.4　Python 分析矢量数据


视频讲解

### 12.4.1　访问矢量数据

Shapefile 文件对于 GIS 数据交换和 GIS 分析来说是一种基础数据格式，对于 Shapefile 文件的编辑和其他操作只需要关注两种类型即可，即.shp 和.dbf 文件。

用户可以使用 PyShp 打开一个 Shapefile 文件：

```
>>> import shapefile
>>> r = shapefile.Reader("test.shp")
>>> r
<shapefile.Reader instance at 0x00BCB760>
```

在上述代码中首先创建一个 Shapefile 文件读取器对象实例，并且将其赋给了变量 r。需要注意的是，当将文件名作为参数传给读取器时并没有使用文件的扩展名。

一旦打开一个 Shapefile 文件并创建一个读取器对象，就可以获取一些相关的地理空间信息。在下面的示例中将通过读取器对象获取 Shapefile 文件的边框、形状类型和记录总数等信息。

```
>>> r.bbox
[-91.3880485553174, 30.29314882296931, -88.18631833931401, 34.960911386784]
>>> r.shapeType
1
>>> r.numRecords
298
```

### 12.4.2 Shapefile 文件操作

下面介绍使用 PySHp 库访问并编辑 Shapefile 数据的方法。

使用 PyShp 库操作一个 Shapefile 文件的代码如下：

```
import shapefile

#创建并写入 shapefile
#创建一个点类型的名为 test 的 shapefile
w = shapefile.Writer('test', shapeType = 1)

w.field('name', 'C')              #创建字符类型字段 name
w.point(120, 30)                  #空间信息
w.record('Beijing')               #属性信息
w.close()

#读取 shapefile
r = shapefile.Reader('test')
shapes = r.shapes()               #读取空间信息
print(shapes[0].points)

records = r.records()             #读取属性信息
print(records[0].name)

r.close()
```

上述代码引入了 shapefile 库，使用 Reader()函数创建了一个 Shapefile 文件读取器对象，并将其赋给了变量 r。Reader()函数使用一个 Shapefile 文件名作为参数。注意，一个 Shapefile 文件实际上至少包括了同名的.shp、.shx、.dbf 三个主文件(以及.prj、.sbn、.sbx 等扩展文件)，向 Reader()传递参数时仅需要提供文件名即可，不需要附带扩展名。当然，附带任意一种 Shapefile 的扩展名也能照常执行。例如，下列各行代码的执行效果完全一样：

```
>>> r = shapefile.Reader("MS")
>>> r = shapefile.Reader("MS.shp")
>>> r = shapefile.Reader("MS.dbf")
```

### 12.4.3　空间查询

空间查询的涉及面较广，其中最常见的一种查询方式便是点包容性查询(事实上点包容性查询是叠加分析的一个特例，但因为极其常见，将其单列出来讨论)。例如，操作者在地图图面上任意画一个框，需要选中框内的某种点状要素；又如，已有全国范围内的机场点位数据，现仅需要选出江、浙、沪三省市境内的所有机场。该查询的本质是从点要素集中找到位于某个多边形内部的所有点要素，从几何学视角上来看，是判断一个点是否在指定的多边形内部。此问题通常采用光影投射法处理，该方法从测试点创建一条直线并穿过多边形，之后计算其和多边形每条边相交后产生的点的个数，若该数目是偶数，则点在多边形外部；若是奇数，则点在多边形内部。其实现代码如下：

**【例 12-1】** 判断点是否在多边形内部(代码 12-1. py)。

```
def point_in_poly(x, y, poly):
  if (x,y) in poly: return True
  for i in range(len(poly)):
    p1 = None
    p2 = None
    if i == 0:
        p1 = poly[0]
        p2 = poly[1]
    else:
        p1 = poly[i-1]
        p2 = poly[i]
    if p1[1] == p2[1] and p1[1] == y and x > min(p1[0], p2[0]) and x < max(p1[0], p2[0]):
        return True

  n = len(poly)
  inside = False

  p1x, p1y = poly[0]
  for i in range(n+1):
    p2x, p2y = poly[i % n]
    if y > min(p1y, p2y):
        if y <= max(p1y,p2y):
            if x <= max(p1x, p2x):
                if p1y != p2y:
                    xints = (y-p1y) * (p2x-p1x) / (p2y-p1y) + p1x
                if p1x == p2x or x <= xints:
                    inside = not inside

        p1x, p1y = p2x, p2y
  if inside: return True
  return False
```

### 12.4.4 叠加分析

尽管点包容性查询更常见，但地理要素往往并不单以点要素形式出现，空间关系也并不仅仅是包含关系这么简单。例如，现在需要知道全国高速路网中哪些道路跨越了多个省市，哪些道路仅位于单一省份内部，这涉及线状要素与多边形的相交及包含关系；又如，在城市规划管理中需要判断是否有实际建设超越了规划允许范围，侵占了公共用地空间，这涉及多边形要素之间的相交关系。

叠加分析是地理信息系统中判断多个地理要素相互空间关系的主要工具。地理要素之间的空间关系包括相交、接触、穿过、内部、包含、重叠等。某些空间关系可以产生新的几何对象，例如线要素之间的相交可以产生一个点，面要素之间的相交可以产生新的面(交集或并集)，也可以从一个面要素中剪除相交部分，将余下的部分作为新的面要素。OGR 库提供了对矢量数据执行叠加分析的常用方法，如表 12.2 所示。

**表 12.2 OGR 库提供的对矢量数据执行叠加分析的常用方法**

| 函　数 | 功　能 | 参　数 | 返　回 |
|---|---|---|---|
| Intersection() | 相交 | (1) 当前几何对象；<br>(2) 其他几何对象 | 相交部分所形成的几何对象 |
| Union() | 合并 | (1) 当前几何对象；<br>(2) 其他几何对象 | 合并部分所形成的几何对象 |
| Difference() | 求差 | (1) 当前几何对象；<br>(2) 其他几何对象 | 相差部分所形成的几何对象 |
| Intersects() | 判断相交 | (1) 当前几何对象；<br>(2) 其他几何对象 | 如果二者相交，返回 True，否则返回 False |
| Disjoint() | 相离 | (1) 当前几何对象；<br>(2) 其他几何对象 | 如果二者相离，返回 True，否则返回 False |
| Touches() | 相接 | (1) 当前几何对象；<br>(2) 其他几何对象 | 如果二者相接，返回 True，否则返回 False |
| Crosses() | 相交 | (1) 当前几何对象；<br>(2) 其他几何对象 | 如果二者相交，返回 True，否则返回 False |
| Within() | 在内部 | (1) 当前几何对象；<br>(2) 其他几何对象 | 如果当前几何对象在其他几何对象内部，返回 True，否则返回 False |
| Contains() | 包含 | (1) 当前几何对象；<br>(2) 其他几何对象 | 如果二者存在包含关系，返回 True，否则返回 False |
| Overlaps() | 重叠 | (1) 当前几何对象；<br>(2) 其他几何对象 | 如果二者存在重叠关系，返回 True，否则返回 False |

## 12.5 Python 与遥感


视频讲解

### 12.5.1 访问影像文件

本节介绍如何使用 GDAL 库来访问影像文件。每个数据集包含一个或多个波段，每个波段又包含像素数据和可能的概视图，同时数据集中包含该数据集所采用的地理参考信息。

使用 GDAL 读取影像数据极其简单，核心方法为如下函数：

```
gdal.Open(filename, eAccess)
```

该函数有两个参数，第一个参数必写，需要输入一个文件路径作为目标影像文件；第二个参数可选，为一个常量(gdal.GA_ReadOnly 或者 gdal.GA_Update)，表示以只读方式还是以读写方式打开该文件，考虑到许多文件的驱动只支持以只读方式打开，这里如果没有特殊要求，建议选择只读方式，或者不填写该参数。如果影像文件打开成功，该函数返回一个 GDAL 的 Dataset 对象；如果打开文件失败，该函数将产生一个 Error。

为了说明如何使用 GDAL 读写影像数据，下面用一个例子将 3 个单独的 Landsat TM 波段合并成一个单独的 RGB 影像。

**【例 12-2】** 读取波段并合并影像(代码 12-2.py)。

```
from osgeo import gdal

band1 = '/data/LT51190392010144BJC00/L5119039_03920100524_B10.TIF'
band2 = '/data/LT51190392010144BJC00/L5119039_03920100524_B20.TIF'
band3 = '/data/LT51190392010144BJC00/L5119039_03920100524_B30.TIF'

in_ds = gdal.Open(band1);
in_band = in_ds.GetRasterBand(1)

gtiff_driver = gdal.GetDriverByName('GTiff')
out_ds = gtiff_driver.Create('/data/out/nat_color.tif', in_band.XSize, in_band_YSize,
    3, in_band. DataType)
out_ds.SetProjection(in_ds.GetProjection())
```

```
out_ds.SetGeoTransform(in_ds.GetGeoTransform())

in_data = in_band.ReadAsArray()
out_band = out_ds.GetRasterBand(3)
out_band.WriteArray(in_data)

in_ds = gdal.Open(band2)
out_band = out_ds.GetRasterBand(2)
out_band.WriteArray(in_ds.ReadAsArray())

out_ds.GetRasterBand(1).WriteArray(gdal.Open(band3).ReadAsArray())

out_ds.FlushCache()
for i in range(1, 4):
    out_ds.GetRasterBand(i).ComputeStatistics(False)

out_ds.BuildOverviews('average', [2, 4, 8, 16, 32])
del out_ds
```

该代码首先导入了 gdal 模块，然后通过 gdal.Open()打开了 TM 波段一文件的第一个波段(该文件事实上只有一个波段)。注意，GDAL 的波段从索引 1 开始，而不是从 0 开始，所以打开文件的第一个波段需要给 GetRasterBand()函数传入数字 1。接下来使用驱动对象来创建新的数据集，因此需要找到 GeoTIFF 驱动对象并使用它的 Create()函数。该函数的第 1 个参数为所创建的数据集的路径，第 2、3 个参数分别是新数据集的列数和行数，第 4 个参数是新数据集里的波段数，第 5 个参数是数据类型。

在创建数据集后就可以添加像素值。因为已经有了来自 GeoTIFF 的 Landsat 波段一对象，所以可以将像素值读进 NumPy 数组。由于 Landsat 的波段一是蓝色波段，所以必须将其放置在输出图像的第 3 个波段位置。接下来还需要将红色和绿色的 Landsat 波段添加到数据集中。

之后使用 FlushCache()函数将数据写入磁盘，最后建立数据集的概视图层。

### 12.5.2 影像裁剪

常见的遥感影像通常具有很大的范围，例如一景 Landsat TM 遥感影像的范围为 170～180km，但实际的分析区域往往远小于这个范围，因此用户往往会将其缩小至感兴趣的那个区域影像。为了达到这个目的，最好的做法是根据目标区域的预定

边框对影像进行裁剪,可以使用 Shapefile 文件作为边框定义文件并将边框之外的数据移除。图 12.1 是包含杭州市边界线的 Landsat TM 影像的截图。



图 12.1　杭州市边界线的 Landsat TM 影像截图

为了对图片进行裁剪,需要执行如下步骤:

(1) 使用 gdal_array 将图片作为数组载入。

(2) 使用 PySh p 库创建一个 Shapefile 文件读取器。

(3) 栅格化 Shapefile 文件使之成为一张包含地理参照信息的图片。

(4) 将 Shapefile 文件影像作为过滤器只获取 Shapefile 文件边框内的影像像素值。

(5) 使用上述掩膜图片对遥感影像过滤。

(6) 删除不在边框范围内的遥感影像数据。

(7) 将裁剪后的遥感影像保存为独立文件。

**【例 12-3】** 影像裁剪(代码 12-3.py)。

```
import operator
from osgeo import gdal, gdal_array, osr
import shapefile
try:
    import Image
    import ImageDraw
```

```
except:
    from PIL import Image, ImageDraw
raster = "stretched.tif"
shp = "hancock"
output = "clip"

def imageToArray(i):
    a = gdal_array.numpy.fromstring(i.tostring(), 'b')
    a.shape = i.im.size[1], i.im.size[0]
    return a

def world2Pixel(geoMatrix, x, y):
    ulX = geoMatrix[0]
    ulY = geoMatrix[3]
    xDist = geoMatrix[1]
    yDist = geoMatrix[5]
    rtnX = geoMatrix[2]
    rtnX = geoMatrix[4]
    pixel = int((x - ulX) / xDist)
    line = int((ulY - y) / abs(yDist))
    return (pixel, line)

srcArray = gdal_array.LoadFile(raster)
srcImage = gdal.Open(raster)
geoTrans = srcImage.GetGeoTrasform()
r = shapefile.Reader("().shp".format(shp))
minX, minY, maxX, maxY = r.bbox
ulX, ulY = world2Pixel(geoTrans, minX, maxY)
lrX, lrY = world2Pixel(geoTrans, maxX, minY)

pxWidth = int(lrX - ulX)
pxHeight = int(lrY - ulY)
clip = srcArray[:, ulY:lrY, ulX:lrX]

geoTrans = list(geoTrans)
geoTrans[0] = minX
geoTrans[3] = maxY

pixels = []
for p in r.shape(0).points:
    pixels.append(world2Pixel(geoTrans, p[0], p[1]))
rasterPoly = Image.new("L", (pxWidth, pxHeight), 1)

rasterize = ImageDraw.Draw(rasterPoly)
rasterize.polygon(pixels, 0)
mask = imageToArray(rasterPoly)
clip = gdal_array.numpy.choose(mask, (clip, 0)).astype(gdal_array.numpy.uint8)
gdal_array.SaveArray(clip, "().tif".format(output), format = "GTiff", prototype = raster)
```

### 12.5.3　重采样

**【例 12-4】**　重采样示例(代码 12-4.py)。

```
import os
from osgeo import gdal

in_ds = gdal.Open('input.tif')
in_band = in_ds.GetRasterBand(1)
out_rows = in_band.YSize * 2
out_columns = in_band.XSize * 2

gtiff_driver = gdal.GetDriverByName('GTiff')
out_ds = gtiff_driver.Create('band1_resampled.tif', out_columns, out_rows)

out_ds.SetProjection(in_ds.GetProjection())
geotransform = list(in_ds.GetGeoTransform())
geotransform[1] /= 2
geotransform[5] /= 2
out_ds.SetGeoTransform(geotransform)

data = in_band.ReadAsArray(buf_xsize=out_columns, buf_ysize=out_rows)
out_band = out_ds.GetRasterBand(1)
out_band.WriteArray(data)

out_band.FlushCache()
out_band.ComputeStatistics(False)
out_ds.BuildOverviews('average', [2, 4, 8, 16, 32, 64])
del out_ds
```

### 12.5.4　影像分类

遥感图像通过亮度值或像元值的高低差异及空间变化来表示不同地物的差异，这是区分不同图像地物的物理基础。遥感图像分类就是利用计算机通过对遥感图像中各类地物的光谱信息和空间信息进行分析，选择特征，将图像中的每个像元按照某种规则或算法划分为不同的类别，然后获得遥感图像中与实际地物的对应信息，从而实现遥感图像的分类。通常分类方法可以分为两种，即监督分类与非监督分类。监督分类用被确认类别的样本像元去识别其他未知类别像元，其中确认类别像元可通过目视识别或野外调查得到，同时用这些种子类别对判决函数进行训练，之后再用训练好的判决函数对其他待分数据进行分类，使每个像元和训练样本作比较，将其划分

到与其最相似的样本类。非监督分类仅仅依靠图像上不同类别的地物光谱信息进行特征提取,再统计特征的差别来达到分类的目的。本书以非监督分类为例介绍使用 Python 对遥感影像进行分类的方法和过程。

为了执行分类算法,这里引入 Python 的 spectral 模块。该模块主要用于处理高光谱影像数据,并内置了若干常用的监督/非监督分类算法。

**【例 12-5】** 影像分类(代码 12-5. py)。

```
import numpy as np
import spectral
from osgeo import gdal

def make_raster(in_ds, fn, data, data_type, nodata = None):
    driver = gdal.GetDriverByName('GTiff')
    out_ds = driver.Create(fn, in_ds.RasterXSize, in_ds.RasterYSize, 1, data_type)
    out_ds.SetProjection(in_ds.GetProjection())
    out_ds.SetGeoTransform(in_ds.GetGeoTransform())
    out_band = out_ds.GetRasterBand(1)
    if nodata is no None:
        out_band.SetNoDataValue(nodata)
    out_band.WriteArray(data)
    out_band.FlushCache()
    out_band.ComputeStatistics(False)
    return out_ds

def stack_bands(filenames):
    bands = []
    for fn in filenames:
        ds = gdal.Open(fn)
        for i in range(1, ds.RasterCount + 1):
        bands.append(ds.GetRasterBand(i).ReadAsArray())
    return np.dstack(bands)

raster_fns = ['L5119039_03920100524_B10.TIF',
    'L5119039_03920100524_B20.TIF',
    'L5119039_03920100524_B30.TIF',
    'L5119039_03920100524_B40.TIF']
out_fn = 'kmeans_prediction.tif'

data = stack_bands(raster_fns)
classes, centers = spectral.kmeans(data)

ds = gdal.Open(raster_fns[0])
out_ds = make_raster(ds, out_fn, classes, gdal.GDT_Byte)
out_ds.FlushCache()
del out_ds, ds
```

kmeans()函数需要包含预测变量的数组作为参数，同时还可以指定所需的输出集群数、最大迭代数、初始集群等，如果不填写，则默认采用10个集群和20个迭代。

## 12.6　“五水共治”资源地理空间分析综合应用

本节以一个具体的案例讲述如何结合遥感与地理信息系统对地理空间数据进行综合分析。

依据地表水水域环境功能和保护目标，我国的地表水按功能高低分为5类，即Ⅰ、Ⅱ、Ⅲ、Ⅳ、Ⅴ，其中Ⅴ类水主要适用于农业用水区及一般景观要求水域，水质恶劣，不能作为饮用水源，而劣Ⅴ类主要指污染程度已超过Ⅴ类的水。随着高空间、高光谱分辨率卫星传感器的不断涌现和完善，针对城市复杂水体的遥感动态监测已成为可能。利用高分遥感数据监测城市劣Ⅴ类小微水体，可用于筛查遗漏劣Ⅴ类小微水体，动态监测水体黑臭程度的变化，定量评价劣Ⅴ类小微水体治理成效，对于全面支撑城市劣Ⅴ类小微水体整治工作、落实“水十条”具有十分重要的意义。

本例采用遥感图像处理与矢量数据处理相结合的方式，意图分辨劣Ⅴ类水体的具体位置并给出其空间分布特征。

在数据方面，本例选取了高分1号卫星作为数据源。高分1号(GF-1)卫星是我国自主研制的首颗空间分辨率优于1米的民用光学遥感卫星，于2014年8月19日发射，8月21日首次开机成像并下传数据。该卫星搭载有两台高分辨率2米全色、8米多光谱相机，具有亚米级空间分辨率、高定位精度和快速姿态机动能力等特点(技术指标见表12.3)，有效提升了卫星综合观测效能。高分1号影像数据具备了对内陆河道观测的优势，全色2米的分辨率可提高水质观测精度，对于水质参数的提取分析起到了极其重要的作用。

**表12.3　GF-1卫星有效载荷技术指标**

<table>
<tr><th>载　　荷</th><th>谱　段　号</th><th>谱段范围/μm</th><th>空间分辨率/m</th><th>幅宽/km</th><th>侧摆能力</th></tr>
<tr><td rowspan="5">全色多光谱相机</td><td>1</td><td>0.45～0.52</td><td rowspan="4">8</td><td rowspan="5">45(两台相机组合)</td><td rowspan="5">±35°</td></tr>
<tr><td>2</td><td>0.52～0.59</td></tr>
<tr><td>3</td><td>0.63～0.69</td></tr>
<tr><td>4</td><td>0.77～0.89</td></tr>
<tr><td>5</td><td>0.45～0.90</td><td>2</td></tr>
</table>

利用高分1号卫星影像对劣Ⅴ类小微水体进行识别和提取主要分为以下步骤。

（1）数据前期处理：高分1号数据预处理、NDWI水体提取及掩膜、地面光谱和黑臭水质参数相关建模。

（2）劣Ⅴ类小微水体综合方法提取：将地面模型应用于卫星图像，对遥感反演的CDOM及浑浊度指标进行分析，将黑臭水体识别结果进行叠加，将CDOM与浑浊度遥感识别结果皆为轻度劣Ⅴ类小微水体的水体判定为疑似劣Ⅴ类小微水体。选择典型劣Ⅴ类小微水体进行水质参数测量，构建劣Ⅴ类小微水体遥感识别模型，利用高分卫星影像对劣Ⅴ类小微水体进行遥感信息识别和提取，结合地面验证和其他数据，获得劣Ⅴ类小微水体分布现状。劣Ⅴ类小微水体卫星遥感判别分级指标主要包括CDOM和浑浊度，按照同时满足两个指标进行判断分级标准，指标如表12.4所示。

**表12.4　城市劣Ⅴ类小微水体卫星遥感判别分级标准**

| 特征指标 | 清洁水体 | 轻　度 | 重　度 |
|---|---|---|---|
| 浑浊度(NTU) | 0～15 | 15～20 | 20～25 |
| CDOM($m^{-1}$) | 0～0.0369 | 0.0370～0.0374 | 0.0375～0.0379 |

首先使用Python创建一些常用函数以备后期使用：

```
import gdal
from osgeo import gdal
from osgeo import gdal_array
from osgeo import ogr
try:
  import Image
  import ImageDraw
except:
  from PIL import Image, ImageDraw

def imageToArray(i):
  """ 将 Python 影像库的数组转换成一个 gdal_array 图片"""
  a = gdal_array.numpy.fromstring(i.tostring(), 'b')
  a.shape = i.im.size[1], i.im.size[0]
  return a

def world2Pixel(geoMatrix, x, y):
  """ 使用 GDAL 的 geoMatrix()方法计算地理坐标对应的像素坐标 """
  ulX = geoMatrix[0]
  ulY = geoMatrix[3]
  xDist = geoMatrix[1]
  yDist = geoMatrix[5]
```

```
    rtnX = geoMatrix[2]
    rtnY = geoMatrix[4]
    pixel = int((x - ulX) / xDist)
    line = int((ulY - y) / abs(yDist))
    return (pixel, line)

def copy_geo(array, prototype = None, xoffset = 0, yoffset = 0):
    ds = gdal.Open(gdal_array.GetArrayFilename(array))
    prototype = gdal.Open(prototype)
    gdal_array.CopyDatasetInfo(prototype, ds, xoff = xoffset, yoff = yoffset)
    return ds
```

接下来使用 gdal_array 将影像直接载入 NumPy 数组：

```
source = "source.tif"
target = "ndwi.tif"
srcArray = gdal_array.LoadFile(source)
srcImage = gdal.Open(source)
geoTrans = srcImage.GetGeoTransform()
green = srcArray[1]
ir = srcArray[3]
```

提取出的绿光波段和红外波段能够代入到 NDWI 指数公式中用于计算，该公式执行方程“(绿光-近红外)/(绿光+近红外)”。

```
gdal_array.numpy.seterr(all = "ignore")
ndwi = 1.0 * ((green - ir) / (ir + green + 1.0))
ndwi = gdal_array.numpy.nan_to_num(ndwi)
gtiff = gdal.getDriverByName("GTiff")
gtiff.CreateCopy(target, copy_geo(ndwi, prototype = source, xoffset = ulX, yoffset = ulY))
gtiff = None
```

通过 NDWI 指数可以提取遥感影像中水体的部分，对该部分分别进行浑浊度和 CDOM 的反演，可以得到水体的浑浊度和 CDOM 的具体数值，根据表 12.4 将浑浊度在 10～20 或 CDOM 数值在 0.0375～0.0379 的水体提取出来，判定为劣 V 类水体。

为了可视化劣 V 类水体在空间上的分布，可以使用地理热力图。这里是先将劣 V 类水体存储到 Excel 表中，注意至少留出经度和纬度两个字段，分别命名为 lon 和 lat。

**【例 12-6】** 将劣 V 类水体以热力图形式展现。

```
import numpy as np
import pandas as pd
import seaborn as sns
import folium
import webbrowser
from folium.plugins import HeatMap

posi = pd.read_excel("waterV.xlsx")

num = len(posi)

lat = np.array(posi["lat"][0:num])          #获取纬度值
lon = np.array(posi["lon"][0:num])          #获取经度值

data1 = [[lat[i],lon[i] for i in range(num)]

map_osm = folium.Map(location = [35,110],zoom_start = 5)
HeatMap(data1).add_to(map_osm)              #将热力图添加到前面建立的 map 里

file_path = r"\waterV.html"
map_osm.save(file_path)                     #保存为 HTML 文件

webbrowser.open(file_path)                  #用默认浏览器打开
```

热力图效果如图 12.2 所示。

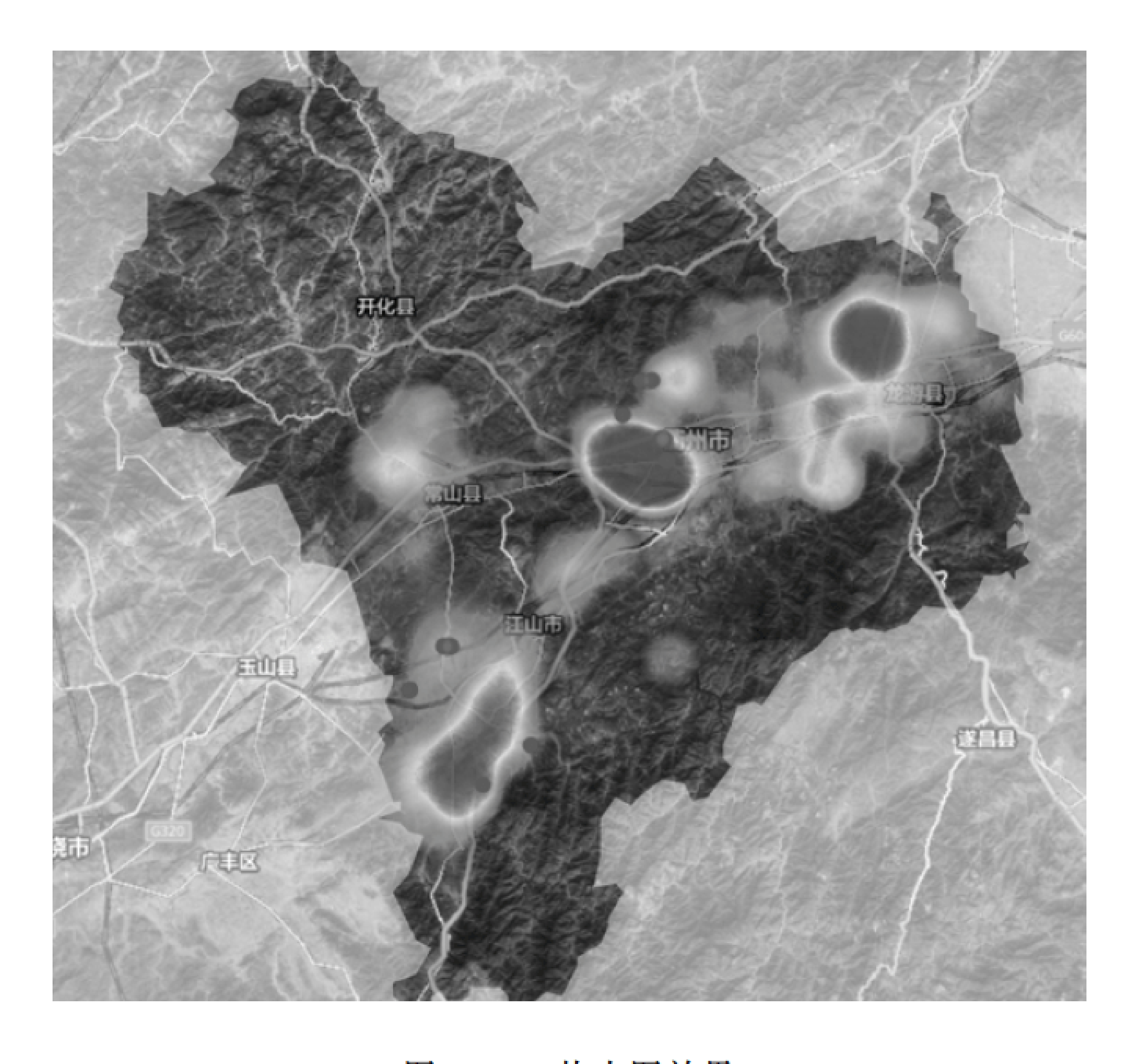



图 12.2 热力图效果

## 本章小结

本章先向读者介绍地理空间分析的基本概念和常用的地理空间数据；然后介绍Python中与地理数据处理和地理分析相关的工具；接着分别介绍Python处理与分析矢量数据和栅格数据的方法；最后以一个具体案例介绍Python地理空间分析的综合应用。

## 习题

1. 矢量数据与栅格数据各有何特征？为何表达空间范围的地物，栅格数据的数据量往往比矢量数据大得多？

2. 编写一个Python函数，输入参数为经度(lon)和纬度(lat)，输出结果为使用GeoJSON格式所表达的点对象。

3. 编写一个Python函数，输入参数为一个shapefile文件，要求在函数中读取该文件并返回该矢量数据的完整属性列表。